# Cloudproxy Nuts and Bolts

John Manferdelli[1]

**Note:** The simpleexample code has been refactored and we've made most of the corresponding changes in this document but we might have missed some we'll complete the update soon.

## Overview

Cloudproxy is a software system that provides *authenticated* isolation, confidentiality and integrity of program code and data for programs even when these programs run on remote computers operated by powerful, and potentially malicious system administrators. Cloudproxy defends against observation or modification of program keys, program code and program data by persons (including system administrators), other programs or networking infrastructure. In the case of the cloud computing model, we would describe this as protection from co-tenants and data center insiders. To achieve this, Cloudproxy uses two components: a "Host System" (raw hardware, Virtual Machine Manager, Operating System) which provides capabilities described below to the protected program or "Hosted System" (VM, Application, Container).

A key concept for Cloudproxy is program measurement. A Host System measures a Hosted System incorporating the actual binary code of the Hosted System and configuration information affecting its execution resulting in a unique, globally descriptive, unforgeable, identity called the "Hosted System Measurement." Since a Host System knows the "measurement" of each Hosted System it runs, it can store secrets that only that Hosted System will receive[2]. The Host System can also "attest" to statements made by Hosted Systems by incorporating the Hosted System Measurement in statements it signs on behalf of the Hosted System. The upshot of this is that a Cloudproxy Hosted System can be isolated, can use secrets only it knows to encrypt and integrity protect all data it receives, sends or stores, as well as having secrets that allow it to securely authenticate itself over an otherwise unprotected network connection. Each Host System can itself be the Hosted System of a parent Host System and each such "child Host System" relies on its parent Host System to protect the keys the child Host System uses to provide services for its Hosted Systems. The "root" or "base" Host System is typically hardware[3] which, fortunately, is widely available.

Learning how to develop Cloudproxy applications is easily achieved by seeing and understanding fairly simple working code. We developed a simple application, called

---

[1] John is manferdelli@google.com or jlmucbmath@gmail.com. Cloudproxy is based on work with Tom Roeder (tmroeder@google.com) and Kevin Walsh(kevin.walsh@holycross.edu) and Fred Schneider.

[2] To do this, the Host System must be isolated and have access to secrets only it knows. Given that, the Host System, using its secrets, simply encrypts and integrity protects the Hosted System secrets along with the Hosted System Measurement. It only decrypts those secrets for a Hosted System with the same Hosted System Measurement.

[3] In our programming example, we use a non-hardware root called the "SoftTao" that is useful for developing and debugging Cloudproxy applications. Hardware root hosts are discussed in [5].

*simpleexample*, that is used throughout this paper. Reading this paper in conjunction with understanding *simpleexample* should enable you to develop Cloudproxy applications within a day or so. The source code for *simpleexample* is in the Cloudproxy distribution.

Readers can consult [1] for a fuller description. Source code for Cloudproxy as well as all the samples and documentation referenced here is in [2]. This paper has a number of appendices, including a high level walk through of the Cloudproxy implementation.

## Principal Names in Cloudproxy

Principal names in Cloudproxy (called "Tao Principal Names") are globally unique, general and can represent key based principals, machine based principals, and, most importantly, program based principals. A Tao Principal Name is hierarchical and, in the case of program principals, contains the Hosted System Measurement as well as the measurement of its Host and all its Host System's ancestors. For example, a principal rooted in a public key will have the cryptographic hash of it in its name making it globally unique and unforgeable. Since a program principal has the Hosted System Measurement in its Tao Principal Name as well as the measurement of the Host System and all its ancestors, it too is globally unique and unforgeable.

Root hosts are key based principals identified by the cryptographic hash of the root's public key (a hardware key like an *AIK* or *Quote Key*, or the key for the SoftTao host). The name for a Hosted System (i.e., a program) running on a "root" Host System includes the Host system key principal name as the first element of the Hosted System name. For example,

```
key([08011001180122450801124l046cdc82f70552eb...]).Program([25fac93bd4c
c868352c78f4d34df6d2747a17f85...])
```

Here, `key([0801100118012245080l1…])` represents the signing key of the root Host System[4] with the hash of the public key in parenthesis. The next element of the Hosted System Principal name is `Program([25fac93bd4cc868352c78f4d34df6d2747a17f85...]`. This represents the Hosted System program; the value in parenthesis for the Program is a cryptographic hash of the code and configuration information of the booted Hosted System[5]. If the Host System were a Linux host rooted in a Trusted Platform Module ("TPM") authenticated boot, its name would include a hash of the PCRs of the booted Linux system, the hash of the Authenticated Code Module ("ACM")[6] that initiated the authenticated boot, the hash of the Linux kernel image and a hash of its initramfs[7]. If the Hosted System was a Linux application, whose Host System was a Cloudproxy enabled Linux hosted by a Hypervisor that booted under a TPM boot, its name

[4] The byte value in key is a SHA-256 cryptographic hash of the corresponding public key serialized in an internal key format. So, if you know the public key, you can verify that the bytes in the key name correspond to that self-same public key.
[5] As indicated by the ellipsis ("…") principal names are often longer and may even contain.
[6] The ACM is a piece of software that controls the authenticated boot of a TPM mediated boot, so it must be included as "configuration information."
[7] Initramfs will have security critical code like the service that implements the Tao so it must be measured along with the kernel image to provide an accurate identity for the "running Linux OS."

would include key based root, the KVM base host as above representing the PCR values and boot flags of the hypervisor at the second layer of the hierarchy, and the measurement (performed by the hypervisor) of the Linux guest OS (again naming its code hash, boot flags and initramfs hash), at the third layer of hierarchy and s the measurement (made by the Linux host) of the application code and configuration information (like command arguments) at the fourth level of the name hierarchy.

In the *simpleexample* execution output, analyzed below, there are many examples of Tao Principal Names.

## The Cloudproxy API

The Cloudproxy programming model is simple and uses only a few API calls. Cloudproxy provides a programming interface in Go or C++ and we refer to the collection of Cloudproxy API's as the "Tao Library."

There are two Cloudproxy principal API interfaces of interest for Go programmers.

The first, and most important API for a developer, is the Tao API which is used to access Host functions in a Hosted System. The interface definition is in `$CLOUDPROXYDIR/go/tao/tao.go`. The interface functions are:

```
GetTaoName()returns the Tao principal name of the Hosted System.

ExtendTaoName(subprin auth.SubPrin irreversibly extends the Tao
Principal Name by adding an additional node to the the hierarchical Tao
Principal Name.

GetRandomBytes(n int) returns n cryptographically secure random bytes.

Rand() returns and io.Reader for random bytes from this Tao.

Attest(issuer *auth.Prin, time, expiration *int64, message auth.Form)
returns an Attestation from the Host System.  The (optional) issuer,
time and expiration will be given default values if nil; the message
attested to is a form in the Tao authorization language.

Seal(data []byte, policy string) returns a blob encrypted and integrity
protected by the Host System containing the data, policy and
measurement of the Hosted System in a form suitable for Unseal.

Unseal(sealed []byte) decrypts and returns the data and policy string,
if access complies with the policy which usually includes having the
embedded Hosted System measurement match the embedded measurement.
```

There are a few additional Tao functions, commonly used by Hosted Systems, related to domain management, and guards employed for authorization decisions:

```
CreateDomain(cfg DomainConfig, configPath string, password []byte)
(*Domain, error)

DomainLoad is used to store and retrieve Program Certificates and
sealed policy data.

Parent()which gets the parent interface to the Tao.  If t :=
tao.Parent(), for example, we'd call Attest as t. Attest(issuer, time,
expiration, message.

DialTLS(network, addr string) creates a new X.509 certs from fresh keys
and dials a given TLS, returns connection.

DialWithKeys(network, addr string, guard tao.Guard, v *tao.Verifier,
keys *tao.Keys) connects to a TLS server using an existing set of keys,
returns net.Conn.

Listen(network, laddr string, config *tls.Config, g tao.Guard, v
*tao.Verifier, del *tao.Attestation) returns a new Tao-based
net.Listener that uses the underlying crypto/tls net.Listener and a
Guard to check whether or not connections are authorized.

ValidatePeerAttestation(a *Attestation, cert *x509.Certificate, guard
Guard) checks a Attestation for a given Listener against an X.509
certificate from a TLS channel.

(l *anonymousListener) Accept() (net.Conn, error) Accept waits for a
connect, accepts it using the underlying Conn and checks the
attestations and the statement.
```

The Tao Library also contains helper functions to build and verify Program Certificates, perform common crypto tasks like key generation and establish the Tao Channel. These are most easily understood by looking at the *simpleexample* code below.

In addition, the Tao Library has rather extensive and flexible authorization support. Authorization decision are performed by *guards*.

Current *guards* include:

- The liberal guard: this guard returns true for every authorization query
- The conservative guard: this guard returns false for every authorization query
- The ACL guard: this guard provides a list of statements that must return true when the guard is queried for these statements.
- The Datalog guard (used in the example below): this guard translates statements in the CloudProxy auth language (see `$CLOUDPROXYDIR/go/tao/auth/doc.go` for details) to datalog statements and uses the Go datalog engine from github.com/kevinawalsh/datalog to answer authorization queries.

A brief description of the guards and authorization language appears in Appendix 1 but you don't need to understand the authorization language to understand *simpleexample*.

Examples of all these calls (and their arguments), except for Rand, appear in the *simpleexample* code.

The second API is the Host API used by utilities to start and stop Hosted Systems and defined in `$CLOUDPROXYDIR/go/tao/host.go`. You will seldom use this API. It consists of the following calls:

```
GetRandomBytes(childSubprin auth.SubPrin, n int) which returns random
bytes

Attest(childSubprin auth.SubPrin, issuer *auth.Prin, time, expiration
*int64, message auth.Form) which requests the Host System's Host sign a
statement on behalf of

Encrypt(data []byte) returns and encrypted, integrity protected blob only
the Host can decrypt.

Decrypt(encrypted []byte) returns the data protected by Encrypt if this
is the right Host.

AddedHostedProgram(childSubprin auth.SubPrin) notifies host that a new
Hosted System has been created.

RemovedHostedProgram(childSubprin auth.SubPrin) notifies the Host that a
Hosted System has been terminated.

HostName() returns the Principal Name of this Host System.
```

The best way to learn Cloudproxy is by examining the annotated code in `$CLOUDPROXYDIR/go/apps/simpleexample` in conjunction with the commentary below.

## The Cloudproxy Tao Paradigm

The Tao is often used in a stereotypical way, which we refer to as the Tao Paradigm. Cloudproxy programs often have policy public keys embedded ($PK_{policy}$) in their image either explicitly or implicitly[8]. Statements signed by the corresponding private key ($pK_{policy}$), and only those statements, are accepted as authoritative and are acted on by these programs. The policy key(s) plus the Hosted System code and configuration, reflected in its measurement, fully describe how the Hosted System should behave and, hence, an authenticated measurement is a reliable description of expected behavior.

[8] Explicitly embedding the key just means that it appears in initialized data measured as part of the program. An example of implicit embedding, which is described in more detail below, is reading in, say, the policy key and extending the identity of the program with that key.

In the Tao Paradigm, when a program first starts on a Hosted System, it makes up a public/private key-pair, called the *ProgramKey*, ($PK_{program}$/ $pK_{program}$) and several symmetric keys that it uses to "seal" information for itself. A Hosted System then "seals," using the Host System interface, all this private (key) information[9]. After that, the Hosted System requests an Attestation from its Host System, naming the newly generated $PK_{program}$ and sends the resulting Attestation to a security domain service which confirms the security properties in the Attestation and Host Certificate[10]. If the Attestation and Host Certificate meet security domain requirements, the security domain service signs (with $pK_{policy}$) an x509 certificate specifying $PK_{program}$ and the Tao Principal Name of the Hosted System. The resulting certificate, called the *Program Certificate*, can be used by any Hosted System to prove its identity to another Hosted System in the same security domain. Program Certificates are used to negotiate encrypted, integrity protected TLS-like channels between Hosted Systems (the "Tao Channel"). A Hosted System can share information over these channels with full assurance of the code identity and security properties of its channel peer. Once established, each endpoint of the Tao Channel "speaks for" its respective Hosted System.

Hosted Systems in the same security domain can fully trust other authenticated Hosted Systems in that security domain with data or processing. Typically, a Hosted System uses the symmetric keys it generates and seals at initialization to encrypt and integrity protect information it stores on disks or remotely.

The Tao Paradigm also often uses a "domain service" to maintain the policies of a distributed Cloudproxy application. Employing a centralized security domain service eliminates the need for each and every Cloudproxy Hosted System in a security domain to maintain lists of trusted hardware or trusted programs and simplifies distribution, maintenance and upgrade.

Often, Hosted Systems in the same security domain will share intermediate keys to protect data that may be used in many Host System environments. These keys can be shared based on policy-key signed directives as Host or Hosted Systems are upgraded or new systems are introduced in a controlled but flexible way eliminating the danger that data might become inaccessible if a particular Cloudproxy system is replaced or becomes damaged or unavailable. The penultimate section of this paper discusses key management techniques as keys, programs and domain information changes over the lifetime of a Cloudproxy based application.

## Hardware roots of Trust

Cloudproxy requires that the lowest level, "booted" system software (the "base system") be measured by a hardware component which provides attest services, seal/unseal services and, usually, some hardware facilities to isolate Hosted Systems. Absent hardware protection, remote users have no principled way to trust the security promises (isolation, confidentiality,

[9] The sample code in `$CLOUDPROXYDIR/go/apps/fileproxy` demonstrates rollback protection for this sealed data and we plan to improve local rollback support in the near future.
[10] The actual attestation being signed by the Host System expressed in a formalized language is $PK_{program}$ speaksfor the Hosted System's Tao Principal Name.

integrity, verified code identity) of Cloudproxy since “insiders” might have silently changed security critical software or stolen low level keys.

Cloudproxy supports TPM 1.2 and TPM 2.0 as hardware roots for Host Systems. We have implemented support for other mechanisms and believe adding a new hardware mechanism is relatively easy. Support for the TPM is limited to Linux systems. Cloudproxy also can initialize and run on a “soft Tao” either on Linux systems or on OS-X systems. This allows for easy development and debugging on a wide variety of platforms.

Once the base Host System is safely measured and booted on supported hardware, Cloudproxy provides for recursive Host Systems at almost every layer of software including:

1. A Host System consisting of hardware (e.g. - TPM, SMX) that hosts a VMM which isolates Hosted Systems consisting of Virtual Machines.
2. A Host System running in an operating system which isolates Hosted Systems consisting of processes (or applications).
3. A Host System running in an operating system which isolates Hosted systems hosted consisting of subordinate Operating Systems or Containers.
4. A Host System running in an application (like a browser) which isolates Hosted Systems consisting of sub-applications, like plug-ins.

In all cases, Hosted Systems employ the same Cloudproxy API and can use any non-Cloudproxy native service (for example, any system call in Linux) so the programming model at each Hosted System layer is essentially unchanged from corresponding non-Cloudproxy code.

## Sample Applications

This paper is intended to allow you to develop and debug Cloudproxy programs rather quickly. To this end we include installation instructions for running under a “soft Tao.” Full instructions for installing and deploying Cloudproxy based systems on TPM 1.2 and TPM 2.0 Intel hardware with SMX extensions appears in a companion paper called “Cloudproxy Nuts and Bolts Deploying Real Cloudproxy Applications” [6]. However, you do not need to read the deployment guide to learn to write Cloudproxy applications. This paper is self-contained.

## Installing Cloudproxy on Linux or a Mac

First, you should download the Cloudproxy repository from [2]. To do this, assuming you have git repository support, type

git clone https://github.com/jlmucb/cloudproxy,

or,

go get https://github.com/jlmucb/cloudproxy.

This latter command will also install the needed go libraries. You can also download a zipped repository from github. You should probably install this in `~/src/github.com/jlmucb` (which

we refer to as `$CLOUDPROXYDIR`) to save go compilation problems later. It's a good idea to put go binaries in `~/bin` as is common in Go. Follow the installation instructions in `$CLOUDPROXYDIR/Doc`; that directory also contains [1] and an up to date version of this document as well as installation instructions for TPM 2.0 capable machines and installation instructions for Cloudproxy enabled KVM hypervisors and Docker containers.

You must also install the Go development tools (and C++ development tools if you use the C++ version). The Go Cloudproxy programs have dependencies on the standard Go packages *protobuf, glog, gtest, crypto, gflags and encoding*, which can be downloaded using the *go get* command) and two additional packages: *github.com/kevinawalsh/datalog* and *github.com/google/go-tpm*, again these can be downloaded using *go get*.

Next, compile, and initialize the *simplexample* application in `$CLOUDPROXYDIR/go/apps/SimpleExample` and run it as described in the next section.

## Understanding Simple Example

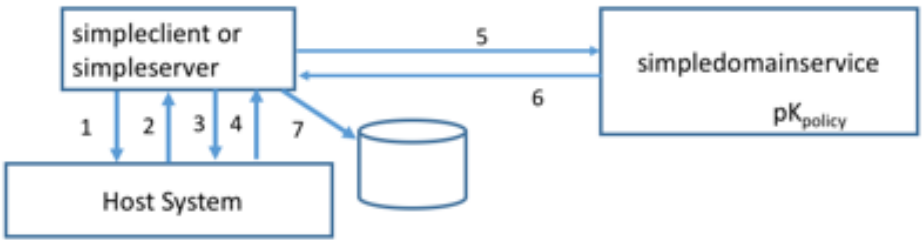
There are three application components in *simpleexample*, each producing a separate executable:

1. A Simple Client in `$CLOUDPROXYDIR/go/apps/SimpleExample/SimpleClient simpleclient.go`.
2. A Simple Server in `$CLOUDPROXYDIR/go/apps/SimpleExample/SimpleServer/simpleserver.go`.
3. A Simple Security Domain Signing Service in `$CLOUDPROXYDIR/go/apps/support_infrastructure/SimpleDomainService/simpledomainservice.go`.

Common code used by the client and server is in `$CLOUDPROXYDIR/go/apps/SimpleExample/common`, this code is specific to example. `$CLOUDPROXYDIR/go/support_libraries/tao_support` and `$CLOUDPROXYDIR/go/support_libraries/domain_policy` have common code for initializing and establishing the tao and interacting with a domain service; this code is fairly general and can be used in many application --- using library code in new or existing applications can can be achieved with very few "cloudproxy specific" additional lines of code.

When *SimpleClient* and *SimpleServer* start for the first time on the Host System, they generate a public/private key pair called a program key. Program Keys authenticate the program. In order to be useful, program keys must be "trusted" by other domain components. This is achieved, in the Tao Paradigm by providing attestations of the program public keys to *simpledomainservice.* If the measurements are correct, *simpledomainservice* signs Program Certificates for those

keys with the private portion of the domain policy key, $pK_{policy}$[11]. This interaction is depicted below.



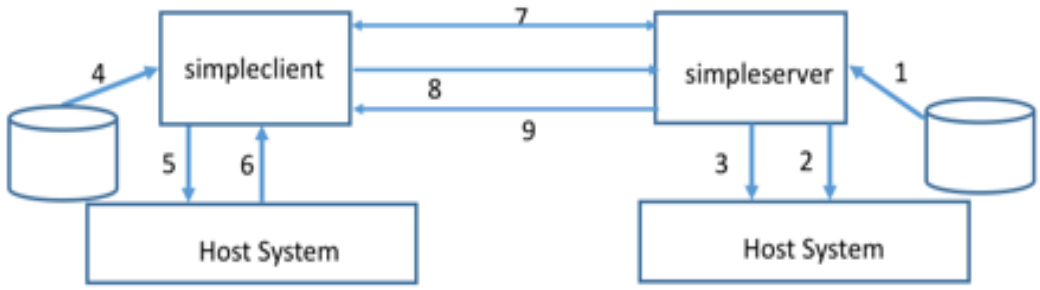

**Initialization**

1. simpleclient (or simpleserver) generates public/private key pair $PK_{simpleclient}$, $pK_{simpleclient}$. simpleclient requests Host System attest $PK_{simpleclient}$.
2. Host System retuns attestation
3. simpleclient generates additional symmetric keys and request Host System seal symmetric keys and $pK_{simpleclient}$.
4. Host System returns sealed blobs.
5. simpleclient connects to simpledomainservice and transmits attestation.
6. simpledomainservice returns signed Program Certificate.
7. simpleclient stores sealed blobs and Program Certificate for later activations.

**Initialization**

After initialization of keys and secrets, *SimpleServer* waits for *SimpleClient* instances to request their secret (each secret is client dependent). Each *SimpleClient* instance uses a Tao Channel to contact the *SimpleServer* to learn its secret. We don't implement rollback protection or distributed key management for intermediate secrets in *simpleexample* just to keep the example as simple as possible. *simpledomainservice* is the domain service for *simpleexample*.

[11] Recall that the policy public key is embedded either explicitly or implicitly in each domain Hosted System and so no CA infrastructure is required.




Operation

1. Simpleserver reads previous sealed blobs and Program Certificate.
2. Simpleserver requests Host System unseal blobs yielding symmetric keys and private program key.
3. Host system returns unsealed blobs.

4. Simpleclient reads previous sealed blobs and Program Certificate.
5. Simpleclient requests Host System unseal blobs yielding symmetric keys and private program key.
6. Host system returns unsealed blobs.

7. Simpleclient and simpleserver open encrypted, integrity protected channel using their program keys and certificates.
8. Simpleclient transmits a request to retrieve secret.
9. Simpleserver retruns secret.

**Operation**

We describe the Cloudproxy API, compilation and installation, execution and output of the Go version of *simpleexample* in the sections below. We also provide a general description of the critical Cloudproxy elements used in the *simpleexample* to help you get used to the programming model. Since the domain service does not use the Cloudproxy API extensively, we don't describe code in detail here although `$CLOUDPROXYDIR/go/infrastructure_support/simpledomainservice` contains a full working version. A corresponding C++ version of *SimpleClient* is described in an appendix.

Although *simpleexample* is in fact very simple, the Tao relevant code in *simpleexample* can be used with little change, even in complex Cloudproxy applications.

### *Simple Client in Go*

*simpleclient* is implemented as a single go file in `$CLOUDPROXYDIR/go/apps/simpleexample/simpleclient/simpleclient.go` using the library code in `$CLOUDPROXYDIR/go/apps/SimpleExample/common`, `$CLOUDPROXYDIR/go/support_libraries/tao_support/taosupport.go` and

`$CLOUDPROXYDIR/go/support_libraries/domain_policy` mentioned above. You should open those files and read the code as you read this.

When it starts, *SimpleClient* parses the flags it was called with and calls `TaoParadigm` (a common support function, in `taosupport.go`, which establishes the "Tao Paradigm". Both *SimpleClient* and *SimpleServer* call `TaoParadigm` with the location of the domain configuration information, the directory in which *SimpleClient* can save its local files (like sealed keys and certs) and a preallocated `TaoProgramData` object. If successful, `TaoParadigm` returns a filled `taosupport.TaoProgramData` object containing the policy cert for the domain and *SimpleClient's* Tao Principal Name, symmetric keys, Program Key and Program Cert. Almost all the Cloudproxy specific code is concentrated in `TaoParadigm` which uses the Tao API; it's described below. Note that *SimpleClient* erases its keys (`defer clientProgramData.ClearTaoProgramData()` after completing the request.

After calling `TaoParadigm`, *SimpleClient* calls `OpenTaoChannel` (another support function in `taosupport.go`) to open up a Tao Channel with *SimpleServer*. If successful, `OpenTaoChannel` returns the encrypted, integrity protected channel stream socket as well as the Tao Principal Name of the *SimpleServer* it connected to.

Finally, *SimpleClient* sends a request for its secret and calls `GetResponse` to retrieve *SimpleServer's* response. If the request was successful, it prints its secret.

### *Simple Server in Go*

*SimpleServer* is implemented as a single go file in `$CLOUDPROXY/go/apps/simpleexample/simpleclient/simpleserver.go` and also employs the support functions mentioned above.

Just as *SimpleClient*, *SimpleServer* parses the flags it was called with and calls `TaoParadigm` with the location of its domain configuration information, the directory in which *SimpleServer* can save its local files and a preallocated `TaoProgramData` object. As above, `TaoParadigm` returns a filled `taosupport.TaoProgramData` containing the policy cert for the domain and SimpleServer's Tao Principal Name, symmetric keys, Program Key and Program Cert.

*SimpleServer* sets up cert chain rooted in the Policy Certificate to validate Program Certificates it received from peer clients, and listens over the indicated socket. In addition to the client Program Certificate, the TLS handshake uses *SimpleServer*'s Program Certificate. After initialization, *SimpleServer* then loops through a standard socket acceptance loop, waiting for successful client connections. Each successful connection is serviced in a separate thread. A successful client connection will result in a valid peer Certificates for the corresponding SimpleClient obtained and verified by the TLS handshake. *SimpleServer* retrieves extracts the *SimpleClient*'s Tao Principal Name from the *SimpleClient* Program Certificate (which appears in

the Organizational Unit of its x509 name) and dispatches a thread [`go serviceThead(ms, clientName, serverProgramData)`] with the per-client service channel, client Principal Name and *SimpleServer's* `TaoProgramData` object.

`serviceThead` loops through a standard service request-response loop. It calls `simpleexample_messages.GetRequest` on the designated service channel to get requests. If it receives a valid request, it calls `HandleServiceRequest` to service the request. There is only one valid request which is "give me my secret," after receiving such a request, `HandleServiceRequest` makes up a client specific secret consisting of the the clientName with "43" appended and returns it to the requesting client using `simpleexample_messages.SendResponse`. After the first successful request, *SimpleServer* terminates.

### *Common support code*

Now we describe the common code used by *SimpleServer* and *SimpleClient* employing the Cloudproxy Tao.

`TaoParadigm` loads the domain information from the provided configuration file [`err := tao.LoadDomain(*cfg, nil)`] and retrieves the policy certificate from the domain information. It calls `simpleDomain.ExtendTaoName(tao.Parent())` to extend the Tao Principal Name with the hash of the public policy key, binding the policy key to the current program identity.

Next `TaoParadigm` calls `LoadProgramKeys`, which retrieves previously created sealed symmetric keys and Program Key along with the Program Certificate (if they exist), from the application directory, otherwise it returns nil. If the sealed sealed symmetric keys were recovered, it unseals them, otherwise (via the call to `InitializeSealedSymmetricKeys`), it generates new keys, seals them and saves them to the correct file in the application directory. If the sealed Program Key exists, it unseals the Program Key, otherwise, `TaoParadigm` generates a new program key, builds a request for *SimpleDomainServer*, attests the new Program public key and transmits the request and attestation to *SimpleDomainServer* to have a Program Certificate signed by the policy key. This is done by `InitializeSealedProgramKey` which we describe further below. Before returning, `InitializeSealedProgramKey`, seals the private Program Key, and stores the sealed key and certificate in the application area and returns the new Program Key and Program Certificate.

Finally, `TaoParadigm` fills the `TaoProgramData` object with the symmetric keys, program key, policy certificate, program certificate and the location of application store; it then returns.

`InitializeSealedProgramKey` carries out the heart of the Cloudproxy key management service, so it and its callees are worth a little further discussion. `InitializeSealedProgramKey` calls `CreateSigningKey` which generates a new ProgramKey and DER encodes the public portion of the key. It then constructs a statement in the authorization language that says "`key(ProgramKey)speaksfor PrincipalName(Program)`". This is called, a delegation statement. `CreateSigningKey` then requests the Host System (via the Tao Interface) attest to the delegation statement. The attestation includes the delegation, the measurement of the Hosted System and the DER encoded public ProgramKey, signed by the Host System. The resulting attestation means "`HostSystem(hash-of-attestation-key) says PrincipalName(hash-of-ProgramKey) speaksfor PrincipalName(Program)`." This attestation along with any relevant supporting certificates, is transmitted to the domain signing service (*simpledomainservice*) via the call to `RequestDomainServiceCert`. The domain service, if the Hosted System measurement conforms to the list of "trusted programs" in the domain, signs the Program Certificate with the (private portion of) the policy key and returns it. The resulting Program Certificate means "`Policy-Key says PrincipalName(Program-Key) speaksfor PrincipalName(Program)`." Any program in the domain receiving the Program Certificate from a communicating program can verify the Program Certificate (using the public portion of the policy) and demand the communicating program "prove possession" of the private portion of the Program Key. Such a proof cryptographically authenticates the communicating program and the Tao properties under which it was created.

Two programs in the security domain, one acting as a client and one acting as a server, use their Program Keys to open an encrypted, mutually authenticated, integrity protected TLS channel (the "Tao Channel"). Once this channel is established, each program knows "the channel speaks for the peer Program Principal." The client side of this channel negotiation is accomplished by the program `OpenTaoChannel`. It simply uses the Program Key of the client (and the received Program Certificate of the server which is authenticated by the policy key) to open the TLS channel. `OpenTaoChannel` returns the resulting bidirectional channel handle and the Tao Principal Name of the server. The corresponding code in the server to open the channel is in *SimpleServer* and uses its Program Key and Program Certificate of *SimpleServer*.

`GetRequest, SendRequest, GetResponse, SendResponse` (in common/messages.go) are simple helper functions to get and send requests and responses. Protect and Unprotect (in taosupport.go) are simple functions to encrypt and decrypt files protected by a Hosted System's symmetric keys.

A Hosted System can also ask the Host System to measure and start other programs. We used a utility (*tao run*, see below) to start *SimpleClient, SimpleServer and simpledomainservice* during initialization, so there was no need to do this in the *sampleexample* code. Starting a Hosted System varies a little depending on the Host System environment. To see how this is done in programmatically Linux, consult `$CLOUDPROXYDIR/go/apps/tao_launch`.

We should mention that *simpleexample* was meant mainly to be instructive (but correct!) so we sometimes repeated code that could have been accessed in the Tao Library for clarity. We also opted for simple, transparent constructions in Go sometimes at the expense of being "idiomatically correct." The `TaoProgramData`, for example, duplicates some data structures in the Tao Library and is defined to simplify and clarify the actual Tao Data but it can be replaced.

We don't describe *simpledomainservice* here since it does not directly call the Tao interface. You can find other example applications in `$CLOUDPROXYDIR/go/apps/demo` and a more complex example in `$CLOUDPROXYDIR/go/apps/newfileproxy`.

An appendix has a brief description of the Datalog policy engine and rules.

## Configuring, compiling and running SimpleExample

When the Tao Host System starts, it requires several kinds of information:

- A public key for the root *Tao* (i.e.- a hardware or a soft-key),
- Host data, including the mechanism used to communicate between the Hosted System and the Host System, rules affecting which Hosted Systems the Host System should run, and, in the case of a hardware rooted Host System, the hardware mechanism that is employed (e.g., TPM 1.2 or TPM 2.0).
- Domain data (in our case for the *simpleexample* domain) including the policy private key, and the self-signed policy cert.

In addition, we need an implementation for the "Host System" which includes support for the Host's isolation mechanism (processes in Linux, VM's in KVM) and information about the channels used for Hosts to communicate with Hosted Systems. In our case, the Host System is Linux and the implementation (whether using a soft tao, TPM 1.2 or TPM 2.0) is *linux_host*[12].

To run simpleexample, do the following. As root

```
mkdir  /Domains
chown yourusername /Domains
```

This creates the storage hierarchy for running the example. As yourusername, type:

```
cd $CLOUDPROXY/go
      go install …
cd $CLOUDPROXY/go/apps/simpleexample/SimpleDomain
./copybins
```

[12] *linux_host* is also the implementation used by a KVM Host System and Docker containers and in fact, all Host Systems running on Linux.

This compiles all the programs and copies the applications into `/Domains`. The run:

```
./initsimpleexample
```

*initsimpleexample* creates an area for the simpleexample data in `/Domains/domain.simpleexample`, copied a template file, *domain_template.simpleexample*, that describes the rules used by Cloudproxy to decide what programs to run and generates a "Soft tao key" which will serve (in the place of a TPM key) as the root Tao. The key is concatenated into `/Domains/domain.simpleexample/domain_template.simpleexample`. Then, as root, run:

```
./initdomain
./runall
```

That's all there is to it. But to help understand the environment set up by these shell scripts and understand the output, let's look a little deeper.

The public key rooting a hardware *Tao* is usually produced by a TPM utility; in the TPM 1.2 nomenclature, this is called the AIK which is accompanied by an "AIK certificate". The public key rooting the TPM 2.0 is a "quote key" which is verified by a TPM protocol using the TPM's the *endorsement* key and accompanying "endorsement certificate." In our demo, we use a "soft Tao" which is rooted in a key generated when the domain was initialized. The key is generated, and appended into *domain_template.simpleexample*, in the shellscript *initsimpleexample* by the call:

```
KEY_NAME="$($BINPATH/tao domain newsoft -soft_pass xxx -config_template
$TEMPLATE $DOMAIN/linux_tao_host)"
echo "host_name: \"$KEY_NAME\"" | tee -a $TEMPLATE
```

"`newsoft`" generates a new root soft key. The arguments following the flags "`-config_template -tao_ -pass`" specify, respectively, the location of the template, the location where the domain information is stored and the password protecting the private policy key. This produces a file containing root Tao key.

*initdomain* sets up some directories. Then, it calls

```
$BINPATH/tao domain init -tao_domain $DOMAIN -config_template $TEMPLATE -
pass xxx
```

This program takes the newly created template (*domain_template.simpleexample*) to perform the following processing: First, it creates a policy public private key pair and saves the private portion (under password protection --- the password in the –pass option) in a file, `/Domains/domain.simpleexample/policy_keys/signer`, and then produces a self-signed policy certificate in `/Domains/domain.simpleexample/policy_keys/signer`. It also produces rules (signed by the policy key) that instructs the Tao host which programs it can run. Since we used the "AllowAll" guard, all programs are run; had we used the datalog, or acl guard, it would have computed the hashes of the files specified in the template and signed rules about those

programs. It also produces a configuration file in `/Domains/domain.simpleexample/tao.config`, that our Cloudproxy host uses for initialization; this file need not be signed since misconfiguration results in incorrect measurements.

The policy cert is a self signed X509 certificate. Here is what it looks like:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1 (0x1)
    Signature Algorithm: ecdsa-with-SHA256
        Issuer: C=US, O=CloudProxy, OU=, ST=WA, CN=SimpleExampleTest
        Validity
            Not Before: Jun  7 20:02:36 2016 GMT
            Not After : Jun  7 20:02:36 2017 GMT
        Subject: C=US, O=CloudProxy, OU=, ST=WA, CN=SimpleExampleTest
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:32:45:e2:78:be:17:85:58:18:64:c5:79:7c:4c:
                    04:e0:25:50:a4:3f:6e:fe:ec:0b:95:bf:c3:3c:bb:
                    6f:9a:ff:14:6e:e4:d2:c6:c0:89:69:37:02:eb:6d:
                    c7:ea:85:c3:fc:d9:c0:90:7e:87:56:cb:a4:db:9b:
                    c5:43:b4:77:61
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Agreement, Certificate Sign
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client
Authentication
            X509v3 Basic Constraints: critical
                CA:TRUE
    Signature Algorithm: ecdsa-with-SHA256
         30:46:02:21:00:bb:21:34:a3:86:73:9a:d5:fd:31:cd:3a:01:
         f6:44:20:5a:8e:39:ac:16:50:91:2d:5a:7c:05:1f:64:e0:9f:
         ae:02:21:00:88:ee:d4:29:dd:14:bb:3b:72:f0:e8:5e:c7:33:
         f2:cd:ff:d1:3d:ae:e3:84:e2:2e:fc:81:55:c3:88:be:64:12
-----BEGIN CERTIFICATE-----
MIIB3jCCAYOgAwIBAgIBATAKBggqhkjOPQQDAjBWMQswCQYDVQQGEwJVUzETMBEG
A1UEChMKQ2xvdWRQcm94eTEJMAcGA1UECxMAMQswCQYDVQQIEwJXQTEaMBgGA1UE
AxMRU2ltcGxlRXhhbXBsZVRlc3QwHhcNMTYwNjA3MjAwMjM2WhcNMTcwNjA3MjAw
MjM2WjBWMQswCQYDVQQGEwJVUzETMBEGA1UEChMKQ2xvdWRQcm94eTEJMAcGA1UE
CxMAMQswCQYDVQQIEwJXQTEaMBgGA1UEAxMRU2ltcGxlRXhhbXBsZVRlc3QwWTAT
BgcqhkjOPQIBBggqhkjOPQMBBwNCAAQyReJ4vheFWBhkxXl8TATgJVCkP27+7AuV
v8M8u2+a/xRu5NLGwIlpNwLrbcfqhcP82cCQfodWy6Tbm8VDtHdho0IwQDAOBgNV
HQ8BAf8EBAMCAowwHQYDVR0lBBYwFAYIKwYBBQUHAwEGCCsGAQUFBwMCMA8GA1Ud
EwEB/wQFMAMBAf8wCgYIKoZIzj0EAwIDSQAwRgIhALshNKOGc5rV/THNOgH2RCBa
jjmsFlCRLVp8BR9k4J+uAiEAiO7UKd0Uuzty8OhexzPyzf/RPa7jhOIu/IFVw4i+
ZBI=
-----END CERTIFICATE-----
```

Next, we come to *runall*. This script runs

```
$BINPATH/tao host init -tao_domain $DOMAIN -hosting process -root -pass
xxx
```

This produces a certificate for the soft Tao root key filling the same role as the "*AIK certificate*" or "*Quote certificate*" in the case of TPM roots. TPM certificates and initialization is discussed in [5]. This certificate is stored in /Domains/domain.simpleexample/ and is transmitted by *SimpleClient* and *SimpleServer* to *SimpleDomainService* when they request Program Certificates.

*runall* sleeps between calls to the utilities to provide time for the resulting files to be written.

Next, runall calls:

```
$BINPATH/tao host start -tao_domain $DOMAIN -host linux_tao_host -pass
xxx &
```

This starts the actual Tao host (*linux_host*).

Next *runall* starts the domain service, *SimpleDomainService*, which receives requests for Program Certificates from programs when they start for the first time on a host to sign their Program Keys using the policy private key (this is why *SimpleDomainService* needs the password protecting the policy private key).

```
$BINPATH/tao run -tao_domain $DOMAIN /Domains/SimpleDomainService -
password xxx &
```

Finally, *runall* calls

```
$BINPATH/tao run -tao_domain $DOMAIN /Domains/SimpleServer -domain_config
"/Domains/domain.simpleexample/tao.config" -path
"/Domains/domain.simpleexample/SimpleServer"
```

and

```
$BINPATH/tao run -tao_domain $DOMAIN /Domains/SimpleClient -domain_config
"/Domains/domain.simpleexample/tao.config" -path
"/Domains/domain.simpleexample/SimpleClient"
```

These programs store application files in the directories `"/Domains/domain.simpleexample/SimpleServer` and `"/Domains/domain.simpleexample/SimpleClient,` respectively; these directories were set up in initdomain. *runallcc* does exactly the same thing except it calls the c++ client, *simpleclient_cc.exe*. You can run *runallcc* instead of *runall* to test the C++ implementation.

The Host stores its data, including configuration information, the Host Certificate (used to validate nested Host System Attestation) and it's sealed keys and data, in `$CLOUDPROXYDIR/go/apps/SimpleExample/SimpleDomain/domain.simpleexample/linux_tao_host`. This is also created in the domain init code. Domain data includes the policy

certificate and corresponding (encrypted) private key, hostname, host type and communications channel used between the Host System and its Hosted Systems, information related to the guards used[13] as well as signatures over the binaries that are part of the domain (if the Host System limits what Hosted Systems it will run) and it's stored in "/Domains/domain.simpleexample/policy_keys.

You will have noticed that all the Cloudproxy setup is accessed via a single utility, called *tao,* which initializes this domain data, activates the *Tao* host and runs the applications.

We provide a standard domain template for *simpleexample*. However, you can generate such a template by running *gentemplate*. This template contains information included in the policy cert, the rules used by the domain when authenticating images and the location of the images which must be measured and recorded in the policy rules.

To rerun *simpleexample*, you will first need to kill previous instances of *linux_host*, *SimpleDomainService*, *SimpleServer* and *SimpleClient* and clear previous application data before re-running *runall*. We provide a script "clean" which deletes the old application files and lock files used by the host (admin_socket) and displays the list of running simpleexample programs. To kill each of these, as root, type:

```
kill -9 pid1, …
```

### What the output from SimpleExample teaches us about the Tao

The most concrete way to understand Cloudproxy is to follow the code example and the output. Here is a brief description of the output of the Go version of *simpleexample* using a "soft" tao.

In our execution setup, recall that the domain information is in `/Domains/domain.simpleexample`; this includes the template, tao prepared configuration files and three directories: `SimpleClient, SimpleServer` and `SimpleDomainService` which are directories in which application information (mostly sealed keys) are stored for, respectively, `SimpleClient, SimpleServer` and `SimpleDomainService`. Binaries are stored in the directory `~/bin` as is customary in go.

After you run *simpleexample*, look at the output. You'll notice, at the beginning:

```
Warning: Passwords on the command line are not secure. Use -pass option
only for testing.
Warning: Passwords on the command line are not secure. Use -pass option
only for testing.
Warning: Passwords on the command line are not secure. Use -pass option
only for testing.
Warning: Passwords on the command line are not secure. Use -pass option
only for testing.
```

[13] Look at `$CLOUDPROXYDIR/go/tao/domain.go` for further details.

```
Linux Tao Service
(key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaa1]).
TrivialGuard("Liberal")) started and waiting for requests
2016/06/18 09:50:25 simpledomainservice: Loaded domain

2016/06/18 09:50:25 simpledomainservice: accepting connections
```

This indicates that the *linux_host, SimpleDomainService, SimpleClient* and *SimpleServer* have been initialized. The second section shows that the *linux_host* for the soft tao (with the indicated key) has started and that it employs the LiberalGuard (i.e. - will run any program). The final line indicates that the domain service is started and waiting for requests.

Next, you'll notice,

```
TaoParadigm: my name is
key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaa1])
.TrivialGuard("Liberal").Program([f421709635 2bfe4508d1e2930373df748d35c
ee8f1efa44ac68d0bc973794063]).PolicyKey([b5548720ac9e56c0de44acbcc69c65
e1b6ac07a833eb297e4f846f643bfd7d4c])
```

This is from *SimpleServer*, and it is the Tao Principal Name of your *SimpleServer* program, running on your Host System, after it has been extended with the hash of the loaded policy key. If you look at the source code for *SimpleServer*, you'll notice that the policy key is not embedded in the code; if it had been, the policy key would be reflected in the program measurement. Instead, we read in the policy key key and extend the *SimpleServer* Tao Principal Name with its cryptographic hash. The Tao Principal Name is hierarchical. The first segment, "`key([c096d85702a63ee1d350f80977...])`", describes the host key; it is the sha-256 hash of the host public attestation key in a canonical internal format. Thus, knowing a host's public key (as you typically would, having it's certificate), you could marshal it[14], hash the marshaled form, and confirm the hash in the Tao name is correct. The second segment, "`TrivialGuard("Liberal")`" describes the policy the host uses to determine what programs it will run; in this case, the "Liberal Guard" will run all programs. In some cases, for example if you "owned" a server, you might have a host which will only run programs signed by you. The third segment, "`Program([f421709...])`", describes the *SimpleServer* program reflecting its measurement. The final segment, "`PolicyKey(f3169d…)`", describes the policy key using the hash of its public key. Observe that the Tao Principal Name fully reflects all the program code as well as the policy it will execute (as represented by the policy key). In our programs, all signed statements chain up to the policy key. This is the principle policy enforcement mechanism in this style of Cloudproxy application.

For the rest of this description, we will simplify terms like "`Program([f421709...])`" as "`Program(`*`program-measurement`*`)`".

[14] The canonical form is the internal format. See SerializeEcdsaKeyToInternalName in domain_policy/domain_policy.go or GetKeyBytes in SimpleExampleCpp/taosupport.cc for examples of how to marshal public keys to hash.

Next, notice the statement:

```
simpledomainservice, speaksfor: key(simpleserver_program_key) speaksfor
key(host-key).Program(simpleserver-measurement).key(policy-key)
```

This is the statement that TaoParadigm will use to request an attestation from the Linux Host System. The resulting Host System supplied attestation is

```
key(hash-of-host-system-key) from notBefore until notAfter says
[simpleserver-program-certificate] speaksfor key(linux-
host).Program(simpleserver-measurement).PolicyKey(hash-of-policy-key)
```

This statement is sent to the domain service which, after checking the measurements and domain policy signs a certificate (with $pK_{policy}$) that includes the statement

```
key(hash-of-policy-key) from notBefore until notAfter says
[simpleserver-program-certificate] speaksfor keyhash-of-linux-host-
key).Program(simpleserver-measurement).PolicyKey(hash-of-policy-key)
```

This is the *SimpleServer* Program certificate. *SimpleServer*, as we described in the code annotations, stores this certificate, and sealed versions of the corresponding private *SimpleServer* ProgramKey and SymmetricKeys. Decrypted and useable versions of these keys are populated in `serverProgramData` by `TaoParadigm`.

After initialization, *SimpleServer* waits for client connections.

*SimpleClient* meanwhile, goes through the same *TaoParadigm* initialization (which is not duplicated here) obtaining its Program certificate. *ServerClient* calls `OpenTaoChannel` with its Program Certificate and corresponding key. You'll notice, later in the output, that *SimpleServer* opens a secure channel with a peer

```
key(hash-of-linux-host).Program(simple-client-
measurement).PolicyKey(hash-of-policy-key)
```

That peer is just your *SimpleClient*. Normally, the Host System on which *SimpleClient* runs will be different from the one *SimpleServer* runs although in our case, they run on the same host.

Finally, you'll notice that *SimpleServer* receives a request

```
2016/02/20 11:27:16     message type: 1
2016/02/20 11:27:16     request_type: SecretRequest
```

and returns the secret which is received by *SimpleClient* as

```
simpleclient: secret is
key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaa1])
.TrivialGuard("Liberal").Program([9711ada89d58b1549a18bc7ed7202b11a6dfc
```

```
3c928a70c8b01b4835685975b73]).PolicyKey([b5548720ac9e56c0de44acbcc69c65
e1b6ac07a833eb297e4f846f643bfd7d4c])43
```

*SimpleClient* encrypts and integrity protects the secret with its symmetric keys and the process concludes.

We have only discussed the major output elements here. Your output will contain much more including log messages from *SimpleDomainServer*.

The Program certificate for the *SimpleClient* (which is in `/Domains/domain.simpleexample/SimpleClient/signerCert`) is:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 0 (0x0)
    Signature Algorithm: ecdsa-with-SHA256
        Issuer: C=US, O=CloudProxy, OU=, ST=WA, CN=SimpleExampleTest
        Validity
            Not Before: Jun 17 23:56:21 2016 GMT
            Not After : Jun 17 23:56:21 2017 GMT
        Subject: C=US,
O=key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaa1]).Trivial
Guard("Liberal").Program([9711ada89d58b1549a18bc7ed7202b11a6dfc3c928a70c8b01b4835
685975b73]).PolicyKey([b5548720ac9e56c0de44acbcc69c65e1b6ac07a833eb297e4f846f643b
fd7d4c]),
OU=key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaa1]).Trivia
lGuard("Liberal").Program([9711ada89d58b1549a18bc7ed7202b11a6dfc3c928a70c8b01b483
5685975b73]).PolicyKey([b5548720ac9e56c0de44acbcc69c65e1b6ac07a833eb297e4f846f643
bfd7d4c]), CN=localhost
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:67:71:56:ab:cd:e5:5c:5e:f3:05:26:6c:43:79:
                    0c:2d:26:c6:d0:6c:eb:aa:06:6b:73:fd:2d:45:b0:
                    e4:77:58:a0:e5:6d:e5:81:25:dc:c1:56:9b:76:01:
                    c2:58:33:a2:5f:fa:da:b2:b3:a7:65:8a:44:ae:14:
                    3d:ad:06:46:47
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Agreement, Certificate Sign
    Signature Algorithm: ecdsa-with-SHA256
         30:44:02:20:6b:82:0b:13:33:d0:bc:b0:e5:af:42:b7:64:88:
         4a:41:1a:14:c8:f9:a1:b6:6e:b8:5c:62:27:63:bb:e9:e3:99:
         02:20:4d:11:51:88:55:3b:ba:fc:25:b3:79:eb:f6:37:46:f3:
         13:1c:94:3f:16:1f:da:09:a3:63:6e:f0:bd:72:86:3a
```

You'll notice that `/Domains/domain.simpleexample/SimpleClient/` also contains the files *sealedsigningKey* (*SimpleClient's* sealed program private key), *retrieved_secret* (the "secret" encrypted with *SimpleClient's* symmetric keys) and *sealedsymmetricKey* (*SimpleClient's* sealed symmetric keys).

The certificate for the root Soft Tao key is:

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 0 (0x0)
    Signature Algorithm: ecdsa-with-SHA256
        Issuer: C=US, O=CloudProxy, OU=, ST=WA, CN=SimpleExampleTest
        Validity
            Not Before: Jul 27 01:49:07 2016 GMT
            Not After : Jul 27 01:49:07 2017 GMT
        Subject: O=Soft Tao Key, CN=Soft Tao Key
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:50:f5:94:36:93:78:c7:c3:a5:94:23:e3:77:f5:
                    a9:47:d4:1f:00:b4:a0:53:d5:82:4a:79:5e:75:90:
                    77:f9:c8:1f:7e:ab:84:49:aa:b5:5f:f5:de:5b:9c:
                    be:61:79:3f:2d:26:93:9c:e8:96:1c:a2:b8:ca:ca:
                    31:1d:dc:f5:99
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Agreement, Certificate Sign
            X509v3 Extended Key Usage:
                TLS Web Server Authentication, TLS Web Client
Authentication
    Signature Algorithm: ecdsa-with-SHA256
         30:44:02:20:73:01:e7:94:3d:0d:5e:6f:e4:38:98:45:e2:d0:
         f3:c2:db:93:d8:e2:5c:f4:f0:aa:15:c0:35:46:b1:af:4a:08:
         02:20:6d:d7:0a:6e:77:72:aa:ce:d6:cf:77:28:c3:9a:da:f6:
         d9:64:71:d4:79:2d:78:68:0f:03:92:03:33:52:35:86
-----BEGIN CERTIFICATE-----
MIIBozCCAUqgAwIBAgIBADAKBggqhkjOPQQDAjBWMQswCQYDVQQGEwJVUzETMBEG
A1UEChMKQ2xvdWRQcm94eTEJMAcGA1UECxMAMQswCQYDVQQIEwJXQTEaMBgGA1UE
AxMRU2ltcGxlRXhhbXBsZVRlc3QwHhcNMTYwNzI3MDE0OTA3WhcNMTcwNzI3MDE0
OTA3WjAuMRUwEwYDVQQKEwxTb2Z0IFRhbyBLZXkxFTATBgNVBAMTDFNvZnQgVGFv
IEtleTBZMBMGByqGSM49AgEGCCqGSM49AwEHA0IABFD1lDaTeMfDpZQj43f1qUfU
HwC0oFPVgkp5XnWQd/nIH36rhEmqtV/13lucvmF5Py0mk5zolhyiuMrKMR3c9Zmj
MTAvMA4GA1UdDwEB/wQEAwICjDAdBgNVHSUEFjAUBggrBgEFBQcDAQYIKwYBBQUH
AwIwCgYIKoZIzj0EAwIDRwAwRAIgcwHnlD0NXm/kOJhF4tDzwtuT2OJc9PCqFcA1
RrGvSggCIG3XCm53cqrO1s93KMOa2vbZZHHUeS14aA8DkgMzUjWG
-----END CERTIFICATE-----
```

### Running SimpleExample on real TPM-enabled machines

Complete instructions on installing and deploying Cloudproxy on real TPM enabled machines using KVM, Linux and Docker are in [5]. You should be able to run *simpleexample* on any of those systems.

### Upgrade and key management scenarios

Since sealed material is only provided to a Hosted System with exactly the same code identity that sealed the material running on the exact same Host System, while isolated by that Host System, you may be worried about lost data when a Hosted System breaks or becomes unavailable or limitations that may affect key management, software upgrade or distribution when the Hosted System runs on other Host Systems. In fact, it is rather easy to accommodate all these circumstances, and many others, efficiently, securely and in most cases automatically using Cloudproxy, *provided* Cloudproxy applications make provisions for this during development.

Below are a few sever example key management techniques that can be used when a Cloudproxy application is upgraded, a new Cloudproxy application (in the same security domain) is launched, or as applications migrate to other Host Systems. All these mechanisms preserve the confidentiality and integrity of all Cloudproxy applications and their data.

There is a discussion of many of the mechanisms, as they might affect client software used across different security domains, by users with no control over the application code while supporting consumer transparency (the most challenging case) in [4]. Here we restrict ourselves to cooperating server applications for simplicity.

To ease description, imagine all application data is stored locally or remotely and probably redundantly in encrypted, integrity protected files. Each file is encrypted and integrity protected with individual file keys and each file key is itself encrypted and integrity protected with a group sealing key. Different groups of file keys are protected by different sealing keys to reduce the risk of universal compromise. Every key has exposed meta data consisting of a globally unique name for the entity it protects, the key type and an "epoch." Epochs increases monotonically as the keys are rotated[15]. As keys for a new epoch become available, the objects they protect are re-encrypted, over a reasonable period of time (the Rotation Period). During this time, keys for the prior epoch are available and can be used to decrypt objects; however, as soon as new epoch keys are available, all new data is encrypted with the new epoch keys. At the end of the Rotation Period, once applications have confirmed that all data is protected with the keys from the most recent epoch, old epoch keys are deprecated.

The first option to deal with "brittle keys" protecting application data is standard: use a distributed key server which authenticates a Tao Program and provisions symmetric keys for data over the authenticated, encrypted, integrity protected Tao Channel. In this case, Cloudproxy applications do not locally store data protection keys[16] but contact a key server (over a Tao Channel). The key server (which does key rotation, etc., as many do) authenticates the Hosted System that needs keys and verifies that it is authorized to receive those keys, and if so, they are transmitted over the Tao Channel. Hosted Systems can be upgraded and all authorization policy can be maintained by the key service in this model. Note that application upgrade is automatic when you use this option even when the policy keys change: New

[15] And you certainly should rotate keys as part of effective cryptographic hygiene!
[16] Although they may cache such keys --- see below.

versions of Hosted Systems simply re-initialize (get new program keys and certificates) using the (centralized or distributed) security domain service and no special provision, aside from current policy at the security domain service, need be provided[17].

Sample code for such a keyserver is `$CLOUDPROXY/go/support_infrastructure/CPSecretService`. Support libraries that allow you to build symmetric key trees (and partitioned secrets) with key rotation support in the style suggested above are in `$CLOUDPROXY/go/support_libraries/protected_objects`, `rotation_support`. This mechanism allows you to cache sealed keys using the key tree to avoid contacting a keyserver on each activation but you will likely want to add a "key change" notification as new key epochs are introduced.

An alternative, less centralized, key rotation mechanism, which we call certificate based key disclosure, allows individual Hosted Systems maintaining their own keys to protect files as well as perform key rotation themselves. When software is upgraded or new programs are introduced, the new programs or upgraded programs come with a certificate signed by the policy key that instruct one Hosted System to disclose these keys to the new version (or new) Hosted System. Since this can result in lost data if a Host System becomes unavailable, Hosted Systems would likely distribute these keys to different instances on different machines to ensure continuity.

Support libraries that implement certificate based key disclosure are in `$CLOUDPROXY/go/support_libraries/secret_disclosure_support`.

Finally, when new data protection keys are established for an application task, Hosted Systems can contact a domain service to receive intermediate keys for registered files or file classes. These keys can be sealed using the Host System provided Seal and used without contacting the service each time the Hosted System starts. This mechanism places additional administrative burden on each Hosted System to contact the "key sharing service" as intermediate keys rotate but this is not uncommon.

It is important to note that while the foregoing descriptions treat keys as "all or nothing" entities, all these scheme have corresponding "split key" implementations to achieve higher security. In addition, any security domain may elect to have an authorized Cloudproxy Hosted System archive data. Such an archive application, upon which security domain policy confers access to data, can, in the background, archive data to (centralized or distributed) repositories. There are many other possible mechanisms to do key management but these should get you started.

There is a further discussion of key management in a Cloudproxy environment in [5] as it affects deployment of Cloudproxy applications.

[17] Many events may cause such a policy change including a determination that previously trusted hardware elements have been compromised.

## Suggested Exercises

That's all there is to using Cloudproxy. Here are some suggested exercises to complete the training:

1. Write a more complicated set of domain applications; for example, see "`$CLOUDPROXYDIR/go/apps/fileproxy`."
2. After reading the Deployment Nuts and Bolts, boot a Linux Host System on TPM supported hardware using the TPM to root the Linux Tao (see … for instructions) and run *simpleexample* on it.
3. After reading the Deployment Nuts and Bolts, boot a KVM Host System on TPM supported hardware and then run a stacked VM host in a Linux partition (see … for instructions). SimpleExample should run fine in the VM(s) with slight changes to the initialization scripts.
4. Explore the Data log engine (examples?)
5. What happens if you make a modification to `simpleclient.go` and immediately run it on `linux_host` without reinitializing the domain?
6. Write and compile some Datalog rules to do some fancier authorization and try it.
7. Understand how to start Hosted Systems by studying "`$CLOUDPROXYDIR/go/apps/tao_launch`."
8. Correct any errors in this paper or the examples and send the corrections or suggestions to us.
9. Write an awesome Cloudproxy based application and tell us and your friends all about it.
10. Repeat step 9 and have fun!

## References

**[1] Manferdelli, Roeder, Schneider, The CloudProxy Tao for Trusted Computing,** http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-135.pdf.
**[2] CloudProxy Source code,** http://github.com/jlmucb/cloudproxy. Kevin Walsh and Tom Roeder were principal authors of the Go version.
**[3] TCG, TPM specs,**
http://www.trustedcomputinggroup.org/resources/tpm_library_specification
**[4] Beekman, Manferdelli, Wagner,** Attestation Transparency: Building secure Internet services for legacy clients**.** AsiaCCS, 2016.
**[5] Manferdelli, Telang,** Nuts and Bolts of Deploying Real Cloudproxy Applications. Doc directory in [2].

# CloudProxy's Authorization Language

Package *auth* implements Tao authorization and authentication, by defining and implementing a logic for describing principals, their trust relationships, and their beliefs.

The grammar for a formula in the logic is roughly:

```
Form ::= Term [from Time] [until Time] says Form
       | Term speaksfor Term
       | forall TermVar : Form
       | exists TermVar : Form
       | Form implies Form
       | Form or Form or ...
       | Form and Form and ...
       | not Form
       | Pred | false | true
```

Quantification variables range over Terms.

```
TermVar : Identifier
```

Times are integers interpreted as 64-bit Unix timestamps.

```
Time ::= int64
```

Predicates are like boolean-valued pure functions, with a name and zero or more terms as arguments.

```
Pred ::= Identifier(Term, Term, ...)
       | Identifier()
```

Terms are concrete values, like strings, integers, or names of principals.

```
Term ::= Str | Bytes | Int | Prin | PrinTail | TermVar
```

Int can be any Go int. Str is a double-quoted Go string. Bytes is written as pairs of hex digits, optionally separated by whitespace, between square brackets. Bytes can also be written as base64w without whitespace between curly braces.

Principal names specify a key or a tpm, and zero or more extensions to specify a sub-principal of that key.

```
PrinType ::= key | tpm
Prin ::= PrinType(Term)
       | PrinType(Term).PrinExt.PrinExt...
PrinExt ::= Identifier(Term, Term, ...)
          | Identifier()
```

Principal tails represent a sequence of extensions that are not rooted in a principal. They are used to make statements about authorized extensions independent of the root principal. For example, they are used to specify that a given program is authorized to execute on any platform. A PrinTail must be followed by at least one extension.

```
PrinTail ::= ext.PrinExt.PrinExt...
```

Identifiers for predicate and principal extension names and quantification variables are limited to simple ascii printable identifiers, with initial upper-case, and no punctuation except '_':

```
PredName ::= [A-Z][a-zA-Z0-9_]*
ExtName ::= [A-Z][a-zA-Z0-9_]*
```

The keywords used in the above grammar are:

```
from, until, says, speaskfor, forall, exists, implies, or, and, not, false,
true, key
```

The punctuation used are those for strings and byte slices, plus:

```
'(', ')', ',', '.', ':'
```

It is possible to represent nonsensical formulas, so some sanity checking maybe called for. For example, in general:

1. The left operand of Says should be Prin or TermVar, as should both operands of Speaksfor.
2. All TermVar variables should be bound.
3. Conjunctions should have at least one conjunct.
4. Disjunctions should have at least one disjunct.
5. Identifiers should be legal using the above rules.
6. The parameter for key() should be TermVar or Bytes.

Specific applications may impose additional restrictions on the structure of formulas they accept.

All of the above elements have three distinct representations. The first representation is ast-like, with each element represented by an appropriate Go type, e.g. an int, a string, or a struct containing pointers (or interfaces) for child elements. This representation is meant to be easy to programmatically construct, split apart using type switches, rearrange, traverse, etc.

The second representation is textual, which is convenient for humans but isn't canonical and can involve tricky parsing. When parsing elements from text:

Whitespace is ignored between elements (except around the subprincipal dot operator, and before the open paren of a Pred, Prin, or, PrinExt) operator, and before the open parenthesis of a Pred, Prin, or, PrinExt).

For binary operators taking two Forms, the above list shows the productions in order of increasing precedence. In all other cases, operations are parsed left to right. Parenthesis can be used for specifying precedence explicitly.

When pretty-printing elements to text, a single space is used before and after keywords, commas, and colons. Elements can also be pretty-printed with elision, in which case keys and long strings are truncated.

The third representation is an encoded sequence of bytes. This is meant to be compact,

relatively easy to parse, and suitable for passing over sockets, network connections, etc. The encoding format is custom-designed, but is roughly similar to the format used by protobuf.

The encoding we use instead is meant to be conceptually simple, reasonably space efficient, and simple to decode. And unlike most of the other schemes above, strictness rather than flexibility is preferred. For example, when decoding a Form used for authorization, unrecognized fields should not be silently skipped, and unexpected types should not be silently coerced.

Each element is encoded as a type tag followed by encodings for one or more values. The tag is encoded as a plain (i.e. not zig-zag encoded) varint, and it determines the meaning, number, and types of the values. Values are encoded according to their type:

An integer or bool is encoded as plain varint.

A string is encoded as a length (plain varint) followed by raw bytes.

A pointer is encoded the same as a boolean optionally followed by a value.

Variable-length slices (e.g. for conjuncts, disjuncts, predicate arguments) are encoded as a count (plain varint) followed by the encoding for each element.

An embedded struct or interface is encoded as a tag and encoded value.

Differences from protobuf: Our tags carry implicit type information. In protobuf, the low 3 bits of each tag carries an explicit type marker. That allows protobuf to skip over unrecognized fields (not a design goal for us). It also means protobuf can only handle 15 unique tags before overflowing to 2 byte encodings.

Our tags describe both the meaning and the type of all enclosed values, and we use tags only when the meaning or type can vary (i.e. for interface types). Protobuf uses tags for every enclosed value, and those tags also carry type information. Protobuf is more efficient when there are many optional fields. For us, nearly all fields are required.

Enclosed values in our encoding must appear in order. Protobuf values can appear in any order. Protobuf encodings can concatenated, truncated, etc., all non-features for us.

Note: In most cases, a tag appears only when the type would be ambiguous, i.e. when encodingTerm or Form.

The Go code that manipulates auth structures is in `$CLOUDPROXY/go/tao/auth`. Code for the acl guard is in `$CLOUDPROXY/go/tao/acl*.go`. The Datalog guard is in `github.com/kevinawalsh/datalog`.

# Simple Domain Protocol

The Domain Protocol is used in the transmission of a Hosted System's attestation which includes the TaoName of the requesting client (like `key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaa1]).TrivialGuard("Liberal").Program([e458cd8a843494d1388f44b7a0da5bbd17b14abae73b0b61ed06f1428c989399]).PolicyKey([b5548720ac9e56c0de44acbcc69c65e1b6ac07a833eb297e4f846f643bfd7d4c])`) to *simpledomainservice*. Here is a brief description of the protocol:

1. A client (*SimpleServer, SimpleClient, simpleclient_cc.exe*) sends the Domain Service a `DomainCertRequest` which includes a key type, a DER encoded PKIX public key description of its program key along with delegation signed by the host that says:
   key(hash-of-host-key-in-internal-key-format) says key(hash-of-program-key-in-internal-key-format) speaks for current-tao-program-name.
2. *SimpleDomainService* gets the public-key from the DER encoded key description and converts it into the internal serialized key form and hashes it using SHA-256. It then confirms the hash of the program key is the same as in the corresponding delegation. It checks the host key is trusted and the program-name corresponds to an approved program. Using the key information extracted from the delegation and the DER encoded key description it signs the program certificate with the policy-key. This certificate as above, has the Tao Program Name in the organizational unit of the subject name. We do not need a delegation but if we did, it would be:
   `key(hash-of-policy-key-in-internal-key-format) says key(hash-of-program-key-in-internal-key-format) speaksfor Tao-Program-name`
3. It returns the signed certificate in a `DomainCertResponse`.

All Tao Principal component names are well specified and can be calculated (and hence confirmed) using just a standard hash function. If someone wanted to calculate these offline with another program, the "specification" is simple, new key types can be easily accommodated and Certificate validation is standard. Key names and delegations are always compact, the name lengths don't depend on the key types at all, even for a quantum public key system.

## SimpleExample in C++

You can also write your CloudProxy protected programs in C++. This section describes how to do it using the code example in SimpleClientCpp which implements the very same functionality the go *SimpleClient* does but in C++.

### Installing C++ libraries

Before compiling and linking your C++ program, you must compile the standard Cloudproxy C++ libraries required. To do this:

```
cd $CLOUDPROXYDIR/src
cmake .
make
make install
```

This will produce, among other things, the `libtao.a`, which implements the authorization support routines we need, `libauth.a`, which implements the Cloudproxy authorization support we need and two helper libraries, `libmodp.a` and `libchromium.a`, needed for a C++ based Cloudproxy program.

`libauth.a` itself, is compiled from code generated (based on a Go implementation) by:

```
go run ${CMAKE_SOURCE_DIR}/../go/apps/genauth/genauth/genauth.go -
ast_file ${CMAKE_SOURCE_DIR}/../go/tao/auth/ast.go -binary_file
${CMAKE_SOURCE_DIR}/../go/tao/auth/binary.go -header_file
${CMAKE_BINARY_DIR}/auth.h -impl_file ${CMAKE_BINARY_DIR}/auth.cc
```

The automated build procedure above sometimes fails, especially on Macs, so we have also provided a shell script `$CLOUDPROXY/src/standalonelibrarybuild.sh`, which also builds these libraries and, assuming you've set up the simpleexample directory structure, puts the libraries in `/Domains` an all needed includes in `/Domains/includes`. The shell script is easier to debug if things go wrong.

You will need to link `libtao.a`, `libmodp.a`, `libauth.a` and `libchromium.a` in your C++ Cloudproxy program. You will have to include ".`pb.h`" files and compile corresponding ".pb.cc" files produced by `protoc`. To produce these files, you win need to copy `$CLOUDPROXY/go/tao/proto/{attestation.proto, ca.proto, datalog_guard.proto and acl_guard.proto}` as well as `CLOUDPROXY/go/apps/simpleexample/domain_policy/domain_policy.proto` into the SimpleClientCpp source directory and run `protoc –cpp_out=. *.proto` on them to obtain the right `*.pb.h` and `*.pb.cc` files. Finally, you will need to copy the directory `/Domains/simpleexample/policy_keys` into `/Domains/simpleexample/SimpleClientCpp/policy_keys` before running the C++ *simpleexample* programs.

You'll see examples of all this in the *SimpleClientCpp* example code and makefiles. You will also need C++ versions of the files

**SimpleClient in C++**

We only implement *SimpleClient* in C++. The reader can make the obvious changes to implement a corresponding.

The C++ *SimpleClient* source code is in `$CLOUDPROXY/go/apps/simpleexample/SimpleClientCpp` and is structured in the same way as the Go version. As for the go code, most of the Tao related support code is collected into a central support library implemented in `taosupport.cc` and an associated include file `taosupport.h` in `$CLOUDPROXY/src/support_libraries/tao_support`; this corresponds to taosupport.go in the Go version. The main loop is in `simpleclient_cc.cc`. The main loop is nearly self-explanatory: it simply
calls support functions in taosupport to create a Tao program data object (as in go), initialize the program data object (using `InitTao` in taosupport.cc), open the TaoChannel (using `OpenTaoChannel` in taosupport.cc), sends a request for its secret (using `SendRequest` in `messages.cc`), receiving the response (using `GetRequest` in `messages.cc`) and finally printing out the retrieved secret.

**Running simpleclient_cc**

First compile simpleclient_cc.exe by typing

```
make -f simpleclient.mak
```

This *makefile* puts the executable in `/Domains`.

We will use the same *SimpleServer* and *simpledomainservice* we used for the Go version of *simpleexample* and simply substitute *simpleclient_cc.exe* for *SimpleClient* when we run the programs. As with the go version, we provide scripts to run the programs:

```
#do the following ONLY if you haven't run the Go
# version in this session.
cd $CLOUDPROXY/go/apps/simpleexample/SimpleDomain
./gentemplate
./initdomainstorage
./initkey
sudo bash
# First make sure previous hosts were killed and the admin_socket
# was freed.
# Now run the programs
  ./runallcc
```

The resulting output should be similar to the Go output and for the same reasons! Here is an example:

```
DomainRequest
2016/06/18 09:50:49 DomainRequest
IsAuthenticationValid
2016/06/18 09:50:49 simpledomainservice, IsAuthenticationValid name is
key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaa1]).T
rivialGuard("Liberal").Program([e458cd8a843494d1388f44b7a0da5bbd17b14abae
73b0b61ed06f1428c989399]).PolicyKey([b5548720ac9e56c0de44acbcc69c65e1b6ac
07a833eb297e4f846f643bfd7d4c])
2016/06/18 09:50:49 simpledomainservice, IsAuthenticationValid returning
true

SimpleDomainService: key principal:
key([a073a070f2263eb17dc60c0b3c1b9769e141222e3a84bd35392b69e6268ac3d6]),
program principal:
key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaa1]).T
rivialGuard("Liberal").Program([e458cd8a843494d1388f44b7a0da5bbd17b14abae
73b0b61ed06f1428c989399]).PolicyKey([b5548720ac9e56c0de44acbcc69c65e1b6ac
07a833eb297e4f846f643bfd7d4c])

simpleclientcpp: Simple client name:
key(c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaa1).Tri
vialGuard().Program(e458cd8a843494d1388f44b7a0da5bbd17b14abae73b0b61ed06f
1428c989399).PolicyKey(b5548720ac9e56c0de44acbcc69c65e1b6ac07a833eb297e4f
846f643bfd7d4c)

2016/06/18 09:50:49 server, peer client name:
key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaa1]).T
rivialGuard("Liberal").Program([e458cd8a843494d1388f44b7a0da5bbd17b14abae
73b0b61ed06f1428c989399]).PolicyKey([b5548720ac9e56c0de44acbcc69c65e1b6ac
07a833eb297e4f846f643bfd7d4c])
2016/06/18 09:50:49 server: at accept
simpleclient: established Tao Channel with
Sent request
2016/06/18 09:50:49 serviceThread, got message:
2016/06/18 09:50:49 Message
2016/06/18 09:50:49    message type: 1
2016/06/18 09:50:49    request_type: SecretRequest
2016/06/18 09:50:49    data:
2016/06/18 09:50:49
2016/06/18 09:50:49 HandleServiceRequest response buffer:
2016/06/18 09:50:49 Message
2016/06/18 09:50:49    message type: 2

2016/06/18 09:50:49    request_type: SecretRequest
simpleclient: secret is
key([c096d85702a63ee1d350f80977163baf507272ed450ec6fc8a7a7837402bcaa1]).T
rivialGuard("Liberal").Program([e458cd8a843494d1388f44b7a0da5bbd17b14abae
73b0b61ed06f1428c989399]).PolicyKey([b5548720ac9e56c0de44acbcc69c65e1b6ac
07a833eb297e4f846f643bfd7d4c])43, done
simpleserver: client thread terminating
```

# Cloudproxy C++ support libraries

As we saw in the last appendix, developing and running a C++ Cloudproxy program requires using four libraries `libmodp.a, libchromium.a, libauth.a and libtao.a`. Here's what they do and how they are built:

**libmodp.a** is a relatively simple library which implements base64 encoding. The code for this library is in `$CLOUDPROXY/src/third_party/modp`.

**libchromium.a** is another simple support library that provides helpers for string manipulation and file name manipulation. The code for this library is in `$CLOUDPROXY/src/third_party/chromium/base`.

**libauth.a** is the library that has support for constructing, interpreting, marshalling and unmarshalling statements in the Cloudproxy Authorization library. For go, these functions are in `$CLOUDPROXY/go/tao/auth/binary.go` and `$CLOUDPROXY/go/tao/auth/ast.go`. The library employs the C++ files `auth.cc` and `auth.h`. These files are produced by the *genauth* utility which reads `binary.go` and `ast.go`. and writes `auth.cc` and `auth.h`. This mechanism allows us to maintain the syntax of the auth language primitives in one place. Improvements to the go versions are "automatically" reflected in the C++ version.
The code for *genauth* is in `$CLOUDPROXY/go/apps/genauth` which is compiled when you do go install … .

**libtao.a** implements the actual Tao interface in C++. This consists mainly of function stubs for `GetTaoName, ExtendTaoName, GetRandomBytes, Attest, Seal` and `Unseal`. These stubs marshal and forward the calls over an rpc channel (also in libtao.a) to `linux_host` and retrieve and unmarshal the results. libtao also contains support for simple sealing policies. The code implementing libtao is in `$CLOUDPROXY/src/tao`.

As mentioned in the prior appendix, all of these libraries should be built when you do a cmake and make in the `$CLOUDPROXY/src` directory. You will need to be careful in your application makefiles to ensure that the directories containing the libraries are in your LIBPATH and that the include files are in the include path. You will also need to build the C++ compiled protobuf files `(*.pb.h and *.pb.cc)` required. In our example, we copied the libs into `/Domains` but this is not required.

Sometimes the automated make commands to build the C++ libraries in `$CLOUDPROXY/src` do not work properly (particularly om Macs) so we have provided a manual make script to build these libraries and move the includes in `$CLOUDPROXY/src/standalonelibrarybuild.sh`, however, please try to use the standard cmake and make in `$CLOUDPROXY/src` as described in the previous section before resorting to this script.

# Cloudproxy code walk-through

## Overview

In this appendix, we give a rough, high level introduction to the Cloudproxy implementation. It is not necessary to understand this to write Cloudproxy programs but it may be useful as a guide for those who want to "look under the hood."

The Cloudproxy source code is divided into three functional areas: initialization, support and utility code (most are run on the domain administrator's machine which is assumed to be secure), the Tao Host code, and a set of libraries which is used by a Hosted System to communicate with its host to obtain *Tao* services (like StartHostedProgram, Seal, Unseal, Attest) along with some support functions (like creating and interpreting authorization statements and performing cryptographic functions).

The initialization, support and utility code sets up domain information which includes generating and storing the policy key and the policy certificate, constructing and signing domain rules, signing the hardware public keys used by its hosts (in the case of TPM 1.2 this is the AIK, in the case of TPM 2.0 this is the Quote key. It also contains a utility that launches Cloudproxy Hosted Systems (this does not run on the domain administrators machine) and some TPM utilities.

The Tao Hosted System library, which is compiled into a Hosted System in addition to providing cryptographic and authorization support also provides the rpc channel implementation used by a Hosted System to communicate with its Host.

The largest of the three bodies of code provides the Host implementations for Soft, TPM1.2, TPM 2.0, and stacked host Taos, the host side RPC for communicating with Hosted Systems and utilities, an interface to hardware roots like TPM 1.2 and TPM 2.0, facilities to build and interpret Tao Principal Names and other auth language based components, an implementation for Attest, Seal, Unseal, and GetRandom and code to create and manage domain information consisting of keys, certificates and domain rules. In addition, the the host implementation must isolate, measure and start its Hosted Systems. A Host runs as a separate privileged process in Linux called *linux_host*.

This division of code is somewhat arbitrary since there is a lot of code used by all three of the functional code units.

Almost all of Cloudproxy is implemented in Go which is a (largely) type safe language. Go also has nice compact independent cryptographic support libraries. There are two non-Go based components: there are a number of support libraries written in C++ that allow you to write Hosted Systems in C++ (but you need to incorporate additional libraries like openssl) and a few utilities.

## Communications utilities

The Tao employs channels for communication between the Host process (*linux_host*) and its Hosted Systems. Similar channels support communications between a utility (*tao_launch*) and the Host process to start and stop Hosted Systems. The transport for these channels is a unix socket. Hosts may also communicate to programs running on other machines, like the domain and attest services but it uses tcp sockets for this purpose.

`$CLOUDPROXY/go/util` contains utilities used to implement channels, routine I/O and message streams. The files are:

```
chanpair.go
log.go
peercred_darwin.go
fd.go
messages.go
pair.go
peercred_linux.go
fdmessagestream.go
messagestream.go
protorpc/protorpc.go
protorpc/protorpc.proto
flags.go
oob_unix_conn.go
path.go
unix_single_rwc.go
```

path.go contains I/O and path stuff. You'll notice two versions of peercerd, one is for OSX (darwin) and one is for linux. The channels support peer authentication in linux and OSX so Hosts can determine user ids and run Hosted Systems under the appropriate UID.

## RPC over Channels for the Tao

The host (usually *linux_host*) provides an interface to its Hosted Systems over the channels described above through a standard rpc mechanism. The implementation for this rpc mechanism is contained in the following files:

```
linux_host_admin_rpc.go
linux_host_admin_rpc_darwin.go
linux_host_admin_rpc_linux.go
linux_host_tao_rpc.go
```

The code employs Go rpc support which includes the notion of a ClientCodec. A ClientCodec implements writing of RPC requests and reading of RPC responses for the client side of an RPC session.

Requests have the form

```
type Request struct {
    ServiceMethod string
    Seq           uint64 // sequence number chosen by client
}
```

The response header has the form

```
type Response struct {
```

```
    ServiceMethod string // echoes that of the Request
    Seq           uint64 // echoes that of the request
    Error         string // error, if any.
}
```

The response is a header written before every RPC return.

RPC clients are created by an rpc call with a communications channel as in `NewLinuxHostAdminClient` below.

```
func NewLinuxHostAdminClient(conn *net.UnixConn) LinuxHostAdminClient {
        oob := util.NewOOBUnixConn(conn)
        c := rpc.NewClientWithCodec(protorpc.NewClientCodec(oob))
        return LinuxHostAdminClient{oob, c}
}
```

Call(serviceMethod string, args interface{}, reply interface{}) is a Go provided function used by clients to invoke named functions over rpc. Corresponding server side Go support, parses calls and dispatches them to the appropriate service handler. This is used, for example, by `StartHostedProgram` to send a request to a Host process to start a new Hosted System:

```
(client LinuxHostAdminClient) StartHostedProgram(spec *HostedProgramSpec)
     …
req := &LinuxHostAdminRPCRequest{
      Path:          proto.String(spec.Path),
      Dir:           proto.String(spec.Dir),
      ContainerArgs: spec.ContainerArgs,
      Args:          spec.Args,
    }
client.oob.ShareFDs(fds...)
err := client.Call("LinuxHost.StartHostedProgram", req, resp)
subprin, err := auth.UnmarshalSubPrin(resp.Child[0].Subprin)
```

The foregoing forms the nucleus of the Host-Hosted System Tao interface.

## Global protobufs

Most critical Cloudproxy data structures with which programs interact are encoded in protobufs (the auth language persistence format is an exception). The critical protobufs are in `$CLOUDPROXY/go/tao/proto`. Here is a brief outline of them.

**datalog_guard.proto** defines protos that describe a datalog rule and a set of rules signed by a key.

**linux_host_admin_rpc.proto** defines protos that describe Linux rpc message formats like LinuxHostAdminRPCRequest, LinuxHostAdminRPCResponse, and LinuxHostAdminRPCHostedProgram.

**acl_guard.proto** defines protos that describe ACL entries and a set of ACL entries signed by a key.

**domain.proto** defines protos that describe Domain configuration information like

**rpc.proto** defines protos that describe rpc channel request and response buffers (RPCRequest, RPCResponse) for Tao service calls.

**attestation.proto** defines protos that describe an Attestation consisting of a public key, a signature, and a statement. The statement is called a delegation and example is: “signer says (issuer from time until exp says message)”.

**keys.proto** defines protos for cryptographic data support including CryptoKey, CryptoAlgorithm enums, a CryptoKeySet consisting of a CryptoKey and a attestation. Standard curves and parameters, ECDSA, RSA key structures, and Cipher mode structures including EncryptedData formats that contain IV, size, context and ciphertext.

**tpm_tao.proto** defines protos that describe Sealed Data formats (HybridSealedData consisting of a SealedKey and EncryptedData).

**linux_host.proto** defines protos that describe Linux host configuration information.

## Initialization utilities and domain information

All initialization is done through a single application called tao which is implemented in `$CLOUDPROXY/go/apps/tao/tao.go`. Tao is run as root. It is very simple. It creates a log, checks arguments and switches on the argument identifying the initialization request as follows:

```
switch cmd {
   case "help":
           help()
   case "domain":
           subcmd(cmd, "tao_admin", args)
   case "host":
           subcmd(cmd, "linux_host", args)
   case "run", "list", "stop", "kill":
           subcmd(cmd, "tao_launch", args)
   default:
           options.Usage("Unrecognized tao command: %s", cmd)
   }
```

*domain* processes domain initialization including generating policy keys and domain rules by calling *tao_admin*.

*host* starts a *linux_host* by calling `linux_host`.

*run, list, stop, and kill* call `tao_launch` which communicates with a running host (`linux_host`) to run a new Hosted System, list all its Hosted Systems, or stop or kill a running Hosted Systems. `subcmd` does a syscall to `tao_launch` or `tao_admin`.

In *simpleexample*, *tao* was called by the scripts `initkey, initdomain, inithost` and `runds` using the arguments:

```
tao domain init -tao_domain $DOMAIN -config_template $TEMPLATE -
```

`pub_domain_address "127.0.0.1" -pass xxx`

This caused *domain.simpleexample_template* to be read and generates the policy_key and other domain info.

```
tao domain newsoft -soft_pass xxx -config_template $TEMPLATE
$DOMAIN/linux_tao_host
```

This caused the SoftTao key to be generated (it is then appended to *domain.simpleexample_template*).

```
tao domain policy -add_host -add_programs -add_linux_host -add_guard -
tao_domain $DOMAIN -pass xxx -config_template $TEMPLATE
```

This causes the named programs to be measured and signed by the policy key to produce rules for the domain determing which programs it can run.

`sudo tao host start -tao_domain $DOMAIN -host linux_tao_host/ -pass xxx &`

This started the linux_host.

`sudo tao run -tao_domain $DOMAIN /Domains/SimpleDomainService &`

This started SimpleDomainService as a Hosted System.

The hosted programs run as a linux user. Linux provides isolation based on user id. For example, if *SimpleServer* should run as *simple_server_account*, we'd start it, logged in as *simple_server_account,* by saying:

```
tao run -tao_domain $DOMAIN /Domains/simpleserver.
```

The code for *tao_admin* is in `$CLOUDPROXY/go/apps/tao_admin`, it starts with a switch on the needed functionality.

```
switch cmd {
        case "help":
                help()
        case "newsoft":
                createSoftTaoKeys()
        case "init":
                createDomain()
        case "policy":
                managePolicy()
        case "user":
                createUserKeys()
        case "principal":
                outputPrincipal()
        default:
                options.Usage("Unrecognized command: %s", cmd)
        }
```

`createSoftTaoKeys` creates the SoftTao key. The nucleus of this processing is:

```
pwd := getKey("soft tao key password", "soft_pass")
tao.NewOnDiskPBEKeys(tao.Signing|tao.Crypting|tao.Deriving, pwd,
keypath, tao.NewX509Name(dt.Config.X509Info))
```

`NewOnDiskPBEKeys` actually generates the key and `NewX509Name` creates an X509 based

structure which is hashed to produce the root Principal Name for this type of Tao.

CreateDomain initializes the Domain information. The nucleus of this processing is:

```
createDomain()
     dt := template()
     dt.Config.DomainInfo.GetPolicyKeysPath()
     getKey("domain policy key password", "pass")
     domain.Config.DomainInfo.GetGuardType()

     err = domain.Save()
     …
     name := makeTPMPrin(tpmPath, aikFile string, pcrNums []int)
```

This is for TPM based domains. Note the call to `makeTPMPrin` which constructs the Tao Principal name for the root Tao. CreateDomain will also initialize Guard based on the template and call `addProgramRules` which signs domain policy naming the programs that t can run. Within the Host subsequent calls to `domain.Guard.Query(query)` will tell the host whether a program (within the `query`) can be run.

managePolicy can add additional programs to the list of those authorized to run by calling `addExecute`.

```
     managePolicy()
         pwd := getKey("domain policy key password", "pass")
         domain, err := tao.LoadDomain(configPath(), pwd)
          addExecute(canExecute, host, domain)
          err := domain.Guard.AddRule(add)
```

*tao_launch* is a utility that communicates with the Tao Host to start Hosted Systems. The code for it is in `$CLOUDPROXY/go/apps/tao_launch`. It starts by creating a connection, creates an Adminclient froming the rpc channel from tao_launch to linux_host.

```
switch cmd {
     case "help":
          help()
     case "run":
          runHosted(&client, flag.Args())
     case "stop":
          for _, s := range flag.Args() {
               var subprin auth.SubPrin
               _, err := fmt.Sscanf(s, "%v", &subprin)
               options.FailIf(err, "Not a subprin: %s", s)
               err = client.StopHostedProgram(subprin)
               options.FailIf(err, "Could not stop %s", s)
          }
        case "kill":
          for _, s := range flag.Args() {
               var subprin auth.SubPrin
               options.FailIf(err, "Not a subprin: %s", s)
               err = client.KillHostedProgram(subprin)
               options.FailIf(err, "Could not kill %s", s)
          }
        case "list":
           names, pids, err := client.ListHostedPrograms()
```

```
            options.FailIf(err, "Can't list hosted programs")
            for i, p := range pids {
                fmt.Printf("pid=%d subprin=%v\n", p, names[i])
            }
            fmt.Printf("%d hosted programs\n", len(pids))
          default:
            options.Usage("Unrecognized command: %s", cmd)
          }
    sockPath := path.Join(hostPath(), "admin_socket")
    conn, err := net.DialUnix("unix", nil, &net.UnixAddr{
                Name: sockPath,
                Net: "unix"})
    client := tao.NewLinuxHostAdminClient(conn)client :=
    tao.NewLinuxHostAdminClient(conn)
```

After creating the client that implements the rpc, if the function is run, it calls runHosted.

```
        spec := new(tao.HostedProgramSpec)
        if process
             binary := util.FindExecutable(args[0], dirs)
        if docker or KVM
             pidOut, err = os.Create(pidfile)
             tty := isCtty(int(os.Stdin.Fd()))
             subprin, pid, err := client.StartHostedProgram(spec)
             s, _ := client.WaitHostedProgram(pid, subprin)
             hostPath = "linux_tao_host
       if stop
                client.StopHostedProgram(subprin)
```

Other functions are imple,ented in an analogous manner.

The corresponding code to start a Host is in `$CLOUDPROXY/go/apps/host/host.go`. Again it switches on the indicated function:

```
    switch cmd {
       case "help":
              help()
       case "init":
              initHost(domain)
       case "show":
              showHost(domain)
       case "start":
              startHost(domain)
       case "stop", "shutdown":
              stopHost(domain)
       default:
              options.Usage("Unrecognized command: %s", cmd)
       }
```

`startHost` below loads the host and opens the service port. `loadHost` loads the host.

```
    startHost(domain *tao.Domain)
           host, err := loadHost(domain, cfg)
           sock, err := net.ListenUnix("unix", uaddr)

    loadHost(domain *tao.Domain, cfg *tao.LinuxHostConfig) (*tao.LinuxHost,
    error)
        if tc.HostChannelType != "tpm" && tc.HostChannelType != "tpm2" {
            tc.HostSpec = cfg.GetParentSpec()
```

```
            if tc.HostSpec == "" {
            options.Usage("Must supply parent_spec for non-TPM stacked
    hosts")
            }
        } else if tc.HostChannelType == "tpm" {
            // For stacked hosts on a TPM, we also need info from domain
    config
            if domain.Config.TpmInfo == nil {
            options.Usage("Must provide TPM configuration in the domain to
    use a TPM")
            }
            tc.TPMAIKPath = path.Join(domainPath(),
    domain.Config.TpmInfo.GetAikPath())
            tc.TPMPCRs = domain.Config.TpmInfo.GetPcrs()
            tc.TPMDevice = domain.Config.TpmInfo.GetTpmPath()
        } else if tc.HostChannelType == "tpm2" {
            // For stacked hosts on a TPM2, we need info from domain config
            if domain.Config.Tpm2Info == nil {
              …
            }
            tc.TPM2InfoDir = domainPath()
            tc.TPM2PCRs = domain.Config.Tpm2Info.GetTpm2Pcrs()
            tc.TPM2Device = domain.Config.Tpm2Info.GetTpm2Device()
        }
    tao.NewStackedLinuxHost(hostPath(), domain.Guard,
         tao.ParentFromConfig(tc), childFactory)
```

After the initialization programs have run, you'll see the following directory structure under `/Domains/domain.simpleexample`.

`/Domains/domain.simpleexample/rules.` This contains the domain rules.

`/Domains/domain.simpleexample/linux_tao_host`

`admin_socket ---` This is a lock file indicating the host socket is initialized,

`cert` This is the root linux host certificate.

`host.config ---` This is a file containing Linux host information (see below).

`Keys ---` This contains the encrypted host keys.

`/Domains/domain.simpleexample/policy_keys ---` This directory contains the policy cert and encrypted key. The key can only be decrypted on the admin machine (for example, the machine that runs *simpledomainservice* in *simpleexample*.

`/Domains/domain.simpleexample/rules/tao.config` This file contains top leve domain information that was used to create the Domain including information about the root tao implementation, policy keys, guard type and domain rules (see below).

host_config

```
type: "root"
hosting: "process"
```

tao.config:

```
domain_info: <
  name: "SimpleExample"
  policy_keys_path: "policy_keys"
```

```
  guard_type: "AllowAll"
>
x509_info: <
  common_name: "SimpleExampleTest"
  country: "US"
  state: "WA"
  organization: "CloudProxy"
>
acl_guard_info: <
  signed_acls_path: "acls"
>
datalog_guard_info: <
  signed_rules_path: "rules"
>
tpm_info: <
  tpm_path: "/dev/tpm0"
  aik_path: "aikblob"
  pcrs: "17,18"
>
```

Initializing and signing the tpm-AIK is an offline process and uses the signing utility *aiksigner*.

Initializing the tpm2-quotekey is done online. It requires an endorsement certificate which is signed by the policy key offline running the utility *Endorsement*. The quote key is created (see tpm2_tao.go) and used in a self-attestation. A TPM2 protocol (ActivateCredential) is used to verify the quote key corresponds to one bound to a legitimate TPM 2.0. The protocol used a QuoteServer ($CLOUDPROXY/go/apps/QuoteServer) which validates the quote key, signs the quote key certificate and encrypts it so that only the TPM which originated the request can decrypt it.

## Domains

The implementation of domain management is in `$CLOUDPROXY/go/util/Domain.go`. The critical domain information is contained in the following structures.

```
type Domain struct {
        Config     DomainConfig
        ConfigPath string
        Keys       *Keys
        Guard      Guard
}

message DomainDetails {
  optional string name = 1;
  optional string policy_keys_path = 2;
  optional string guard_type = 3;
  optional string guard_network = 4;
  optional string guard_address = 5;
  optional int64 guard_ttl = 6;
}

message DomainConfig {
  optional DomainDetails domain_info = 1;
  optional X509Details x509_info = 2;
```

```
    optional ACLGuardDetails acl_guard_info = 3;
    optional DatalogGuardDetails datalog_guard_info = 4;
    optional TPMDetails tpm_info = 5;
    optional TPM2Details tpm2_info = 6;
  }

  message DomainTemplate {
    optional DomainConfig config = 1;
    repeated string datalog_rules = 2;
    repeated string acl_rules = 3;

    // The name of the host (used for policy statements)
    optional string host_name = 4;
    optional string host_predicate_name = 5;
    // Program names (as paths to binaries)
    repeated string program_paths = 6;
    optional string program_predicate_name = 7;
    // Container names (as paths to images)
    repeated string container_paths = 8;
    optional string container_predicate_name = 9;
    // VM names (as paths to images)
    repeated string vm_paths = 10;
    optional string vm_predicate_name = 11;
    // LinuxHost names (as paths to images)
    repeated string linux_host_paths = 12;
    optional string linux_host_predicate_name = 13;
    // The name of the predicate to use for trusted guards.
    optional string guard_predicate_name = 14;
    // The name of the predicate to use for trusted TPMs.
    optional string tpm_predicate_name = 15;
    // The name of the predicate to use for trusted OSs.
    optional string os_predicate_name = 16;
    // The name of the predicate to use for trusted TPM2s.
    optional string tpm2_predicate_name = 17;
  }
```

`CreateDomain(cfg DomainConfig, configPath string, password []byte) (*Domain, error) creates` domain keys and guards using `SetDefaults()` which picks reasonable defaults for any unspecified options. The domain info is saved in the directories specified using `Save()`. You'll notice one other function in the file, `func (d *Domain) ExtendTaoName(tao Tao.` ExtendTaoName uses a Domain's policy key to extend the Tao with a subprincipal PolicyKey([...]). When a Tao program reads in a policy key, its Tao Name is extended by this subprincipal.

### TPM interface

The TPM 1.2 interface is in `github.com/jlmucb/go-tpm`. It implements the low-level TPM 1.2 interface to TPM 1.2 which is usually at /dev/tpm0. In the Cloudproxy model, the root host "owns" the TPM for the duration of its lifetime.

### TPM2 interface and TPM2 attest service

The TPM2 interface code is in `$CLOUDPROXY/go/tpm/tpm2`. It implements the low-level TPM 2.0 interface to TPM 2,0 which is usually at /dev/tpm0. The TPM 2.0 interface is different from that of TPM 2.0 with different command formats, more algorithm support and a new credential activation protocol (ActivateCredential, mentioned above). TPM 2.0 is only supported by Linux versiok 4 kernels and higher. The TPM is usually at /dev/tpm0 and as in the TPM 1.2 case, the root host "owns" the TPM for the duration of its lifetime.

### Attestation

Attestation support is in `$CLOUDPROXY/go/tao/attestation`, it uses the attestation proto and makes extensive use of the auth language.

### Host initialization and Tao service operation

Hosts are started as processes in linux by calling *tao_launch* host start command. *linux_host* implements the linux host. The source code for *linux_host* is in the following files:

```
$CLOUDPROXY/go/tao/apps/linux_host
$CLOUDPROXY/go/tao/tao.go
$CLOUDPROXY/go/tao/tpm_tao.go
$CLOUDPROXY/go/tao/tpm2_tao.go
$CLOUDPROXY/go/tao/root_host.go
$CLOUDPROXY/go/tao/Soft_tao.go
$CLOUDPROXY/go/tao/host.go
$CLOUDPROXY/go/tao/stacked_host.go
```

In most cases, *linux_host* implements a Stacked Tao; the sole exception is when it implements a root host for a Soft Tao, based on a key created by `NewRootLinuxHost`. Except for a root host, each `linux_host` will have a host (stored in `cachedHost` in `tao.go`). When the host is a TPM, the host Tao interface that linux_host uses to obtain Tao services from its host, is created by `NewTPMTao` or `NewTPM2Tao` in `tpm_tao.go` and `tpm2_tao.go` respectively. Other stacked hosts will have other *linux_host's* as their Host and are created by calling `NewStackedLinuxHost`. A Config (in `config.go`) stores the information about the Tao, its Host Tao, and the way it creates Hosted Programs.

All clients use the Tao interface below to obtain service. TPM based Tao's, Stacked Tao's and Soft Tao's all support this interface, which is called though the typed interface provider for the service, for example, `func (lh *LinuxHost) GetTaoName(child *LinuxHostChild) auth.Prin`. Here is the interface:

```
type Tao interface {
     GetTaoName() (name auth.Prin, err error)
     ExtendTaoName(subprin auth.SubPrin) error
     GetRandomBytes(n int) (bytes []byte, err error)
     …
     Attest(issuer *auth.Prin, time, expiration *int64, message auth.Form)
          (*Attestation, error)
      Seal(data []byte, policy string) (sealed []byte, err error)
     Unseal(sealed []byte) (data []byte, policy string, err error)
}
```

In `tao.go`, you will notice a function `Parent` which returns the clients Tao interface (by returning `cachedHost`).

Initialization proceeds as described below occurs when *linux_host* is started. After initialization, Tao services, for its Hosted Systems, are obtained through the RPC server in the host which dispatched requests to the appropriate service function based on RPC request from its Hosted Systems, and returns the result in the RPC response. A *HostedProgramFactory* manages the creation of hosted programs and requires different implementations at different levels of the software stack (hypervisor, OS/VM, or container). For example, in Linux, it might create processes using fork, or it might create processes running on docker containers. It might also start a virtual machine containing a new instance of an operating system.

When *linux_host* starts, `main` in `go/apps/host/host.go` calls `LoadDomain()` to load domain information and then calls `startHost(domain, cfg)` with the configuration information. `startHost` calls `loadHost` which gets the parent host information, for example, the TPM certs if the parent is a TPM. In the case of a stackedHost, it retrieves the HostChannelType to initialize the channel to the parent.

```
var childFactory tao.HostedProgramFactory
case tao.ProcessPipe:
childFactory = tao.NewLinuxProcessFactory("pipe", socketPath)
case tao.DockerUnix:
childFactory = tao.NewLinuxDockerContainerFactory(socketPath, rulesPath)
```

NewRootLinuxHost (in `linux_host.go`) creates a new LinuxHost as a root Host that can provide the Tao to hosted Linux processes.

```
tao.NewRootLinuxHost(hostPath(), domain.Guard, pwd, childFactory)
tao.NewStackedLinuxHost(hostPath(), domain.Guard,
tao.ParentFromConfig(tc), childFactory)
```

`NewRootLinuxHost` is pretty simple, it generates keys and then gets a `LinuxHost` from `NewTaoRootHostFromKeys`, it also constructs the host name.

```
func NewRootLinuxHost(path string, guard Guard, password []byte,
childFactory HostedProgramFactory) (*LinuxHost, error) {
lh := &LinuxHost{
     guard:        guard,
     childFactory: childFactory,
}
k, err := NewOnDiskPBEKeys(Signing|Crypting|Deriving, password, path,
nil)
rootHost, err := NewTaoRootHostFromKeys(k)
rootHost.taoHostName =
rootHost.taoHostName.MakeSubprincipal(guard.Subprincipal(
lh.Host = rootHost
```

To start a Hosted System, `linux_host` calls `StartHostedProgram` using the `LinuxHost` structure.

```
func (lh *LinuxHost) StartHostedProgram(spec HostedProgramSpec)
     (auth.SubPrin, int, error)
```

```
            id := lh.nextChildID
            prog, err := lh.childFactory.NewHostedProgram(spec)
            hostName := lh.Host.HostName()
            subprin := prog.Subprin()
            childName := hostName.MakeSubprincipal(subprin)
            if !lh.guard.IsAuthorized(childName, "Execute", []string{}) {
                  error
            }

            channel, err := prog.Start()
            child := &LinuxHostChild{channel, subprin, prog}
            go NewLinuxHostTaoServer(lh, child).Serve(channel)
            pid := child.Cmd.Pid()
            // go wait for exit
            return subprin, pid, nil

            go func() {
                 err = tao.NewLinuxHostAdminServer(host).Serve(sock)
                    }()
            // signal handling
```

Getting information about the parent host during initialization is done by `ParentFromConfig(tc Config) Tao` in `tao.go`. which also obtains a Tao interface to the parent services.

```
func ParentFromConfig(tc Config) Tao
     switch tcEnv.HostChannelType {
     case "tpm":
          aikblob, err := ioutil.ReadFile(tcEnv.TPMAIKPath)
          if err != nil {
               error
          }
          …

         host, err := NewTPMTao(tcEnv.TPMDevice, aikblob, pcrs)
         if err != nil {
               error
         }

         cachedHost = host
     …
      return cachedHost
     }

     func Parent() Tao {
             ParentFromConfig(Config{})
             return cachedHost
     }
```

Note that despite the name (host) `cachedHost` is actually a Tao interface. A root Tao is actually created from the keys by `func NewTaoRootHostFromKeys(k *Keys) (*RootHost, error` in `root_host.go`.

While the Tao interface is used by a Hosted System to obtain services from its host, the Host interface is used by *tao_launch* to call host functions. This interface is in `host.go`. There are only two HostTaoTypes: `Root` and `Stacked`. Because the environment calls Host in response to

requests from hosted processes invoking the Tao interface, several Host methods resemble methods in Tao. Semantics and method signatures differ slightly, however, since the environment can add context (e.g., the sub-principal name of the requesting child) or do part of the implementation (e.g., manage policy on seal/unseal). *linux_host* must implement either a root or stacked host interface.

```
 type Host interface {
     GetRandomBytes(childSubprin auth.SubPrin, n int)
          (bytes []byte, err error)
…

     Attest(childSubprin auth.SubPrin, issuer *auth.Prin,
        time, expiration *int64, message auth.Form) (*Attestation, error)
     Encrypt(data []byte) (encrypted []byte, err error)
     Decrypt(encrypted []byte) (data []byte, err error)

        …
     HostName() auth.Prin
}
```

Once `linux_host` is initialized, it waits to accept connections to start Hosted Systems from running Hosted Systems or *tao_launch*.

## Linux support

Linux host support is implemented in `$CLOUDPROXY/go/tao/linux_host.go`. Each host must have a process factory that implements `HostedProgramFactory`. Three different implementations are provided in `$CLOUDPROXY/go/tao/linux_process_factory.go`, `$CLOUDPROXY/go/tao/kvm_coreos_factory.go` and `$CLOUDPROXY/go/tao/linux_docker_container_factory.go`.

A structure called `LinuxHost`, maintains information about the `linux_host` instance including its host, the path to its data, it's childFactory and information about all its Hosted Programs.

```
type LinuxHost struct {
        Host           Host
        path           string
        guard          Guard
        childFactory   HostedProgramFactory
        hostedPrograms []*LinuxHostChild
        hpm            sync.RWMutex
        nextChildID    uint
        idm            sync.Mutex
}
```

LinuxHostChild holds state associated with a running Hosted System.

```
type LinuxHostChild struct {
        channel      io.ReadWriteCloser
        ChildSubprin auth.SubPrin
        Cmd          HostedProgram
}
```

## Keys

Most of Cloudproxy's crypto is implemented in `$CLOUDPROXY/go/tao/Keys.go`. This code is largely self-explanatory and uses the Go crypto support.

## Auth Language

The Cloudproxy authorization language is implemented in the following files in `$CLOUDPROXY/go/tao/auth`.

```
ast.go
binary.go doc.go
lexer.go
scan.go
string.go
auth_test.go
buffer.go
format.go
parser.go
shortstring.go
```

## Guard interface

The Cloudproxy's guards are implemented in the following files in `$CLOUDPROXY/go/tao/Keys.go`.

```
guard.go
datalog_guard.go
acl_guard.go
```

but they also use the datalog guard implementation in `github.com/kevinawalsh/datalog`.

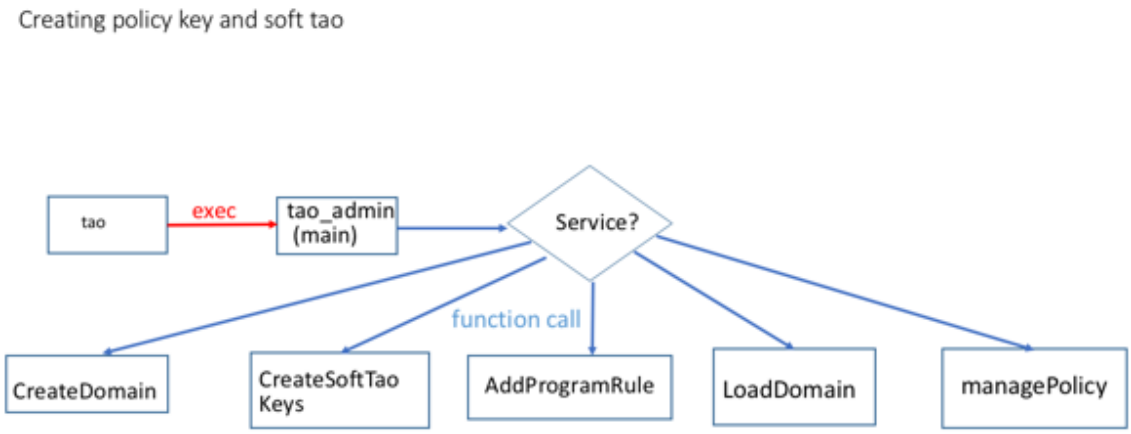
## Miscellaneous support functions

There are additional support functions implemented in the following files:

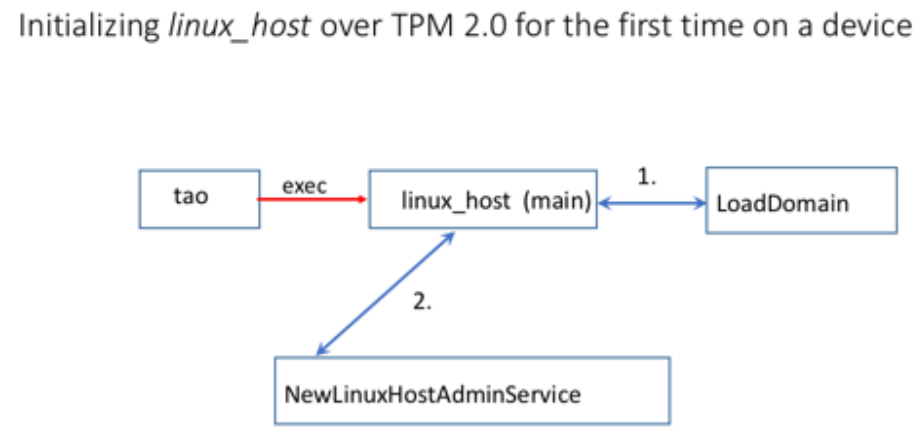
```
client.go
listener.go
errors.go
```

### Call graphs for linux_host

Admin call (generate soft tao key, generate policy key, manage policy, add rule)

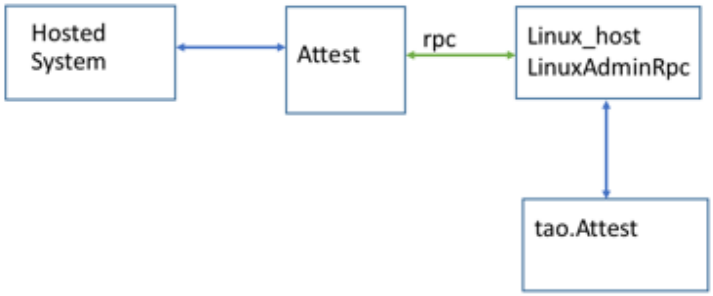



Initialize TPM 2.0 Linux host for the first time on HW



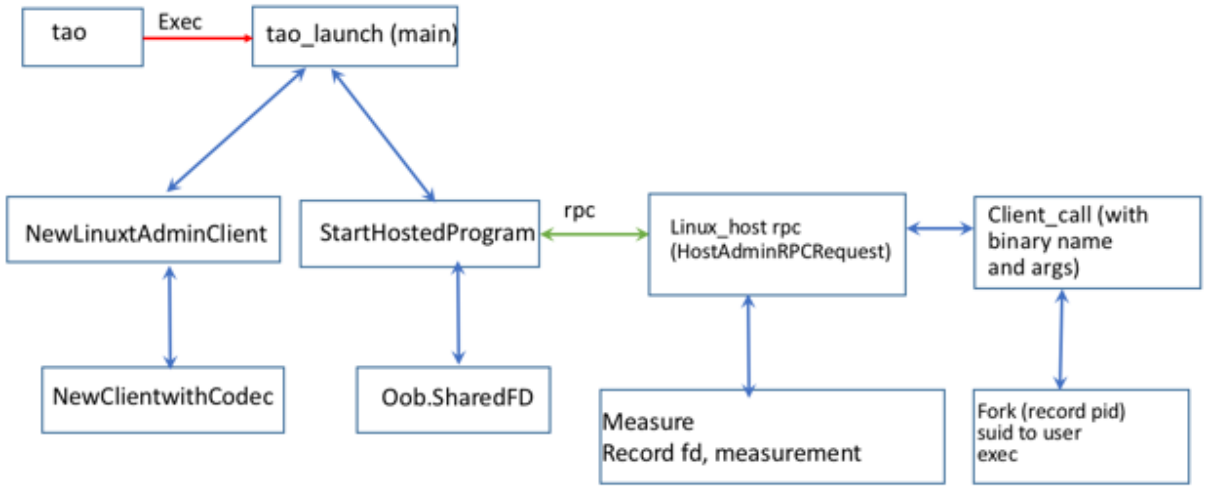

Call a host service (attest)

Making a *tao* interface call (attest) from a Hosted System, consisting of a linux process.




Start a new hosted system

Starting a new Hosted System, consisting of a linux process, under a running *linux_host*.




**Termination**

**C++ Client Library**

The C++ library support is implemented in `$CLOUDPROXY/src/auth.[cc, h]` and `$CLOUDPROXY/src/apps/encode_auth.cc` and the following files in `$CLOUDPROXY/src/tao`:

```
fd_message_channel.h
tao_rpc.cc
util.cc
tao_rpc.h
util.h
message_channel.cc
tao_rpc.pb.cc
message_channel.h
fd_message_channel.cc
tao.h
tao_rpc.proto
```

auth.cc and auth.h are actually generated by a go program in the directory `$CLOUDPROXY/go/genauth`, consisting of `cppgen.go`, `genauth/genauth.go`, and `visitors.go`.

# Rollback

After the first version of this document was completed, we added local rollback protection. We demonstrated remote rollback protection in the original prototype [1] but it relied on network connectivity. Local rollback protection is implemented as follows.

A monotonic counter is maintained for each rollback protected sealed secret. The monotonic counter is identified (uniquely) by the Tao Principal name of the program generating and using the secret and a "label" distinguishing the secret from others used by the program. Each Host, maintains a table of these monotonic counters for its Hosted Systems. This table is encrypted and integrity protected with keys which the Host, in turn seals using the rollback protection mechanism of its Tao Host. At the lowest level Tao, instead of a table, the monotonic counter is part of the hardware "root of trust." Currently, only TPM 2.0 (and the Soft Tao, for testing) root hosts are supported.

Each time a rollback protected seal is executed, the related counter is bumped. The host seals (using a conventional seal operation), the following proto structure:

```
// This is the entry used by the host to track the stored counter value.
message rollback_entry {
  required string hosted_program_name = 1;
  required string entry_label = 2;
  optional int64 counter = 3;
}
```

After a pre-determined number of counters are updated by a Host for its Hosted Systems, the Host generates new keys, encrypts and integrity protects the rollback table and saves it. The Host then saves a rollback protected and sealed version of these new keys.

The programming interface for local rollback protection consists of four functions:

```
(func (tao *tao) InitCounter(label string, counter int64)(error) …)
(func (tao *tao) GetCounter(label string) (int64, error) …)
(func (tao *tao) RollbackProtectedSeal(label string, dataToSeal []byte,
policy string)([]byte, error)…)
(func (tao *tao) RollbackProtectedUnseal(sealedBlob []byte) ([]byte, string,
error)…)
```

These functions do the following:

`InitCounter` initializes a counter identified by label.
`GetCounter` returns the current counter identified by label.
`RollbackProtectedSeal` returns a sealed blob for the secret identified by label.
`RollbackProtectedUnseal` returns unsealed blob and policy corresponding to the sealed

blob input, if no rollback is detected.

Each `RollbackProtectedSeal` increments the counter for the secret identified by label. `RollbackProtectedUnseal` unseals the blob and compares the current counter identified by label to the value of that counter contained in the sealed blob. If they do not match, it returns an error, otherwise, it returns the secret.

Note that in the event a `RollbackProtectedUnseal` fails, one could Unseal the rollback blob, deserialize the marshalled proto and determine the secret, in the event one wishes a less stringent rollback failure enforcement policy.

The corresponding C++ interface is:

```
bool tao.InitCounter(string& label, int64 initial_counter)
bool tao.GetCounter(string& label, int64_t* counter)
bool tao.RollbackProtectedSeal(string& label, string& data_to_seal, string&
policy, string* sealed_output)
bool tao.RollbackProtectedUnseal(string& sealed_data, string* recovered_data,
string* policy)
```

SimpleClient and SimpleClientCpp in simpleexample have sample code for all these interfaces.

Users should be aware of a subtlety in the TPM 2.0 monotonic counter. Because of initialization requirements, the counter is treated as having value floor[(value_returned_byGetCounter + 1)/2]. When a `RollbackProtectedSeal` is performed, the counter is bumped by 2, if the counter is odd and 1, if it is even. The counter is stored with an odd value. When we want to read the counter after start-up, we must bump the counter before the first read. This will make the then-current counter even and we detect startup by noticing it is even. As a result, we must take care to immediately create a new counter epoch and re-encrypt and re-rollback protect the counter table protected by this counter at start-up, because, unless we do this, if we crash before the first rollback seal is performed, we will erroneously detect a rollback condition.